TECHNISCHE
UNIVERSITÄT
DARMSTADT

Automating Type Soundness Proofs for Domain-Specific Languages

vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
von

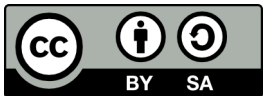
Sylvia Christine Grewe, M.Sc.

geboren in Überlingen, Germany

Referenten der Arbeit:	Prof. Dr. Mira Mezini
	Prof. Dr. Laura Kovács
	Prof. Dr. Sebastian Erdweg
Tag der Einreichung:	29.05.2019
Tag der mündlichen Prüfung:	12.07.2019
Hochschulkennziffer:	D 17

Darmstadt, 2019

URN: urn:nbn:de:tuda-tuprints-90255
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/9025



Abstract

Type systems for static programming languages are supposed to ensure the absence of type errors in code prior to execution. Type systems that meet this expectation are called *sound* type systems in the literature. In practice, however, many type systems are *unsound*, i.e. they successfully type-check programs with type errors which get stuck during execution due to undefined behavior. To reliably ensure that a type system is sound, a sub-area in programming languages research proposes to develop *type soundness proofs*: One proves a soundness property for a logical specification of a type system via logical deduction. The *mechanization* of such a proof shall ensure the absence of human-made deduction errors within the different reasoning steps: A verification system checks that all steps within a proof a correct with regard to the rules within the used logic.

However, developing mechanized type soundness proofs with the tool support and methodologies available today is a cumbersome task even for verification experts: The available support often requires to spell out a large number of "trivial" steps within such proofs manually, which necessitates a certain level of skills and expertise in the area of mechanized verification. Often, language developers and researchers who are experts in conducting type soundness proofs on paper are not necessarily also well-versed in using tool support for mechanized verification. These developers and researchers are typically quickly frustrated by the effort required for mechanized verification and hence often do not attempt it.

The main goal of this thesis is to raise the degree of automation for mechanizing type soundness proofs. To this end, we first study existing mechanization efforts for type soundness proofs from the literature. We use our observations on the one hand to restrict the set of languages we consider in this thesis: We focus on domain-specific languages (DSLs) without first-class binders. On the other hand, we use our observations to identify general shortcomings of existing verification systems regarding how well they support experts in different verification domains in developing domain-specific, automated proof strategies for their domain.

Based on our observations, we propose a generic verification infrastructure called VeriTaS for the automation of domain-specific verification tasks: VeriTaS is a lightweight library in Scala for combining high-level automated domain-specific proof strategies with existing automated theorem provers for the verification of individual, low-level proof steps. VeriTaS is generic in a format for input specifications. Hence, the infrastructure may be instantiated for different verification domains.

We instantiate our VeriTaS verification infrastructure for generating type soundness proofs of DSLs: We provide a domain-specific input format for type system specifications and basic tactics for creating low-level proof steps. Furthermore, we

present automated proof strategies that generate proof structures for type soundness proofs. We evaluate our proof strategies on two case studies, both type systems of representative DSLs. Also, we conduct an empirical study to compare different encoding strategies for low-level proof problems. We used the results of our empirical study to raise the degree of automation provided by our proof strategies for type soundness proofs of DSLs further.

Our case studies show that our instantiation of VeriTaS for type soundness proofs of DSLs achieves a higher degree of automation for such proofs than existing systems.

Zusammenfassung

Typsysteme für statische Programmiersprachen sollen die Abwesenheit von Typfehlern in Quellcode prüfen, bevor ein Programm ausgeführt wird. Typsysteme, die diese Erwartung erfüllen, werden in der Literatur als *korrekte* Typsysteme bezeichnet. In der Praxis sind viele Typsysteme allerdings nicht korrekt, d.h. sie akzeptieren unter Umständen Programme mit Typfehlern, die während der Ausführung zu undefiniertem Verhalten führen. Um verlässlich die Korrektheit eines Typsystems sicherzustellen schlägt ein Untergebiet in der Forschung zu Programmiersprachen vor, *Korrektheitsbeweise von Typsystemen* zu entwickeln: Man beweist formal eine Korrektheitseigenschaft für eine logische Spezifikation eines Typsystems mit Hilfe von logischen Schlussfolgerungen. Die *Mechanisierung* eines solchen Beweises soll sicherstellen, dass der Beweis keine menschengemachten Schlussfolgerungsfehler enthält: Ein Verifikationssystem überprüft, ob alle Schritte in einem Beweis korrekt im Hinblick auf die Regeln der verwendeten Logik sind.

Allerdings ist die Entwicklung von mechanisierten Korrektheitsbeweisen für Typsysteme mit Hilfe heutiger Methodik und heute verfügbaren Werkzeugen selbst für Experten im Bereich der formalen Verifikation eine äußerst aufwendige Aufgabe: Heute verfügbare Werkzeuge erfordern oft, dass eine große Anzahl “trivialer” Beweisschritte manuell ausformuliert werden, was ein gewisses Level an Fähigkeiten und Expertise im Gebiet der formalen Verifikation erfordert. Entwickler von Programmiersprachen sowie Forscher, die Experten darin sind, wie man Korrektheitsbeweise von Typsystemen auf dem Papier durchführt, sind oft nicht gleichzeitig versiert im Umgang mit Werkzeugen für mechanisierte Verifikation. Solche Entwickler und Forscher sind typischerweise schnell frustriert angesichts des nötigen Aufwands für mechanisierte Verifikation und verfolgen daher die Mechanisierung ihrer Beweise oft nicht weiter.

Das Hauptziel dieser Arbeit besteht darin, den Automatisierungsgrad für die Mechanisierung von Korrektheitsbeweisen für Typsysteme zu erhöhen. Hierfür studieren wir zunächst existierende Arbeiten zur Mechanisierung solcher Beweise aus der Literatur. Wir verwenden unsere Beobachtungen hierzu auf der einen Seite, um die Menge der Programmiersprachen, die wir in dieser Arbeit betrachten, einzuschränken: Wir beschränken uns auf domänenspezifische Programmiersprachen (DSLs) ohne vollwertige Konstrukte zum Binden von abstrakten Namen. Auf der anderen Seite verwenden wir unsere Beobachtungen um zu identifizieren, welche generellen Schwächen existierende Verifikationssysteme besitzen, wenn es darum geht, Experten in verschiedenen Verifikationsdomänen darin zu unterstützen, domänenspezifische, automatische Beweisstrategien innerhalb ihrer Domäne zu entwickeln.

Auf der Basis unserer Beobachtungen schlagen wir eine generische Verifikations-

infrastruktur namens VeriTaS für die Automatisierung von domänenspezifischen Verifikationsaufgaben vor: VeriTaS ist eine leichtgewichtige Programmierbibliothek in Scala um automatische, domänenspezifische Beweisstrategien für die Generierung von Hauptschritten von Beweisen mit existierenden automatischen Theorembeweisern zu kombinieren, welche einzelne, technische Beweisschritte verifizieren. VeriTaS ist generisch in einem Eingabeformat für Problemspezifikationen. Daher kann die Infrastruktur für verschiedene Verifikationsdomänen instanziiert werden.

Wir instanziieren unsere VeriTaS-Verifikationsinfrastruktur für die Generierung von Korrektheitsbeweisen für Typsysteme von DSLs: Wir stellen ein domänenspezifisches Eingabeformat für Typsystemspezifikationen sowie Basistaktiken bereit, welche einzelne detaillierte Beweisschritte erstellen. Außerdem präsentieren wir automatische Beweisstrategien zur Generierung von Beweisstrukturen für Korrektheitsbeweise von Typsysteme. Wir evaluieren unsere Beweisstrategien anhand von zwei Fallstudien, beides Typsysteme von repräsentativen DSLs. Zusätzlich führen wir eine empirische Studie durch, um verschiedene Enkodierungsstrategien für Beweisprobleme auf Detailebene miteinander zu vergleichen. Die Ergebnisse unserer Studie haben wir dazu verwendet, den Automatisierungsgrad unserer Beweisstrategien für Korrektheitsbeweise von Typsystemen in VeriTaS weiter zu erhöhen.

Unsere Fallstudien zeigen, dass unsere Instanziierung von VeriTaS für Korrektheitsbeweise von Typsystemen von DSLs einen höheren Automatisierungsgrad für solche Beweise erreicht als existierende Systeme.

Publications

Until the submission of this thesis, we have published contributions contained in this thesis at peer-reviewed journals, conferences and workshops as follows:

- The mechanized type soundness proof in Dafny presented within the survey of existing verification systems in Chapter 3 was published at the Vampire Workshop in 2016 [GEM16]. The remainder of Chapter 3 appears in this thesis for the first time.
- The general research vision for how to automate type soundness proofs of domain-specific languages was published at the "Onward!" conference in 2015 [Gre+15], including an early version of the VeriTaS system proposed in this thesis. The same publication also contained an early version of the typed SQL case study presented in Chapter 3 and in Chapter 7 of this thesis. The author of this thesis presented a refined version of the overall research vision for the VeriTaS verification infrastructure (Chapter 4) at the "SPLASH Doctoral Symposium" in 2016 [Gre16].
- The requirements and design of the VeriTaS verification infrastructure (Chapter 4) and the details of its instantiation for type soundness proofs of DSLs (Chapter 5) were published as a system description paper at the international conference "Principles and Practice of Declarative Programming (PPDP)" in 2018 [Gre+18b].
- The details of how to encode proof problems within an input format for VeriTaS (SPL) into first-order logic (Chapter 5) were first published at the international conference "Principles and Practice of Declarative Programming (PPDP)" in 2016 [Gre+16]. This paper also included a first version of the empirical study presented in Chapter 8 of this thesis. An extended version of the same empirical study was published as a journal paper at the journal "Science of Computer Programming" in 2018 [Gre+18a]. The study within this latter publication is the one presented in Chapter 8. Furthermore, this journal publication contained descriptions of the two case studies that we present in Chapter 7 in this thesis.
- We published early versions and smaller comparison studies of our encoding strategies presented in Chapter 5 at two Vampire workshops ([GEM15] and [GPM18]).

The following further research contributions from the author of this thesis are included in peer-reviewed, published entries in the "Isabelle Archive of Formal Proofs (AFP)" and have not been included in this thesis.

- An Isabelle/HOL Formalization of the Modular Assembly Kit for Security Properties [Bra+18]
- A Formalization of Declassification with WHAT-and-WHERE-Security [Gre+14a]
- A Formalization of Strong Security [Gre+14b]
- A Formalization of Assumptions and Guarantees for Compositional Noninterference [GMS14]

Acknowledgements

Only few people can claim that they achieved anything all on their own - and I am no exception to this rule. Completing this dissertation would not have been possible without the help and support of many people.

My first heartfelt thanks go to my supervisor Mira Mezini, for giving me the opportunity to work on the topic of this dissertation in a research-friendly environment. Thank you, Mira, for receiving me in your group at a time when probably only few people would have taken me in.

Many thanks also go to Laura Kovács, who kindly agreed to act as second referee for this dissertation. Thanks for reading through all these pages and for providing helpful comments via mail during the time I was writing.

I thank Sebastian Erdweg for being my second supervisor: first as a PostDoc in Mira's group, and later remotely on his way to becoming a professor. Thank you for your patience with me, for providing me with an excellent starting point for working on the topic of this dissertation, for improving my work with your feedback, and for working with me on most of the publications that made it into this thesis. And finally, thank you for acting as third referee for this dissertation and for giving me helpful comments on numerous chapters.

I would also like to thank the remaining three members of my PhD committee: Reiner Hähnle, Kristian Kersting, and Stefan Roth.

The people who gave me feedback on parts of this dissertation deserve the next round of thanks: The most dedicated proof-reader was Ragnar Mogk, who helped me to significantly improve several chapters as well as the Introduction. Especially, thank you for spending hours with me re-iterating the first two pages! In this context, also many thanks to my husband Richard Grewe who gave me additional feedback on the first two chapters. I also thank Guido Salvaneschi for proof-reading several chapters in this thesis.

For preparing any complete and readable dissertation, you first need a solid base: in my case, a LaTeX template! I know of many a researcher who spent months in fine-tuning such a template for their theses. I have the unbelievable luck that my husband, Richard Grewe, is a very gifted LaTeX programmer and proud contributor to CTAN. And so of course, he had the perfect LaTeX template for a dissertation already. Richard passed his template along to me as a birthday gift, together with live support. This has been one of the most useful gifts I ever received: It saved me tons of time while preparing this thesis. Thank you a million times for this thoughtful and useful birthday gift!

Next in line are all the people with whom I had the honour to work together and/or discuss together during the last years.

First, there are all the people who were directly involved in shaping the vision of VeriTaS, both the conceptual as well as the technical part (apart from the people mentioned already): Nada Amin, Caspar Bach Poulsen, Oliver Bracevac, Lars Hupel, Sven Keidel, Robbert Krebbers, Edlira Kuci, Ragnar Mogk, Frank Pfenning, Manuel Weiel, and a number of participants from the Oregon Programming Languages Summer School 2015 and at the SPLASH Doctoral Symposium 2016 all discussed and improved with me the general VeriTaS vision. I thank Ragnar and Manuel for discussing VeriTaS' design as well as some concrete code from the Scala implementation with me. In particular, Manuel helped a lot with developing the embedded DSL for writing SPL specifications, and Ragnar provided a lot of help with some tricky database bugfixes.

Next, there are the "ATP and TPTP people", who helped me use automated theorem provers and encode proof problems using TPTP dialects: There is of course the "TPTP guy", Geoff Sutcliffe, with whom I exchanged numerous mails regarding the usage of the TPTP and whom I also met during the Vampire workshops. Thank you for your great work with the TPTP! Next in line are all the "Vampire people": Andrei Voronkov, Laura Kovács (mentioned above already), Martin Suda, Giles Reger, Simon Robillard, and other participants of the Vampire workshops. Thank you for repeatedly having me at the workshops, giving me feedback for my work, and lots of general advice on how to use Vampire for my proof problems. Please do keep up your great work! Apart from the Vampire people, there are the developers of two other ATPs that I tried out in between: Stephan Schulz (eprover) and Philipp Rümmer (princess). Thank you for helping me use your provers and for giving me various insights into the world of automated theorem proving.

I continue with the very important group of all the students who worked with me these past few years as student research assistants or during their Bachelor/Masters theses, and who all helped developing the implementation of various parts of VeriTaS: Michael Raulf and Daniel Lehmann (who both improved early versions of VeriTaS, notably the translation to TPTP, and who helped porting the first version of VeriTaS to Scala), André Pacak (who, most noteworthy, carried out large parts of the empirical study with me and implemented ScalaSPL), and Friedrich Weber (who implemented a proof-of-concept on the generation of auxiliary lemmas for progress and preservation proofs).

A number of people discussed with me selected special topics: I thank Sarah Nadi for helping me with empirical methods and statistics during the preparation of the empirical study from Chapter 8, Dominic Steinhöfel for explaining me key features of KeY, Michael Reif for proving me helpful pointers on call graphs, Sebastian Proksch for general insights on the process of finishing a dissertation, and Reiner Hähnle for general discussions regarding automated theorem proving and KeY.

Moreover, there were a number of people from the Active Group in Tübingen as well as from the Software Technology Group at TUD who listened to trial defense talks and improved the final talk with their feedback. Especially, I would like to thank Mike Sperber from Active Group and Anna-Katharina Wickert from ST for their extensive feedback.

I also thank all my other colleagues from the Software Technology Group and

from the Software Engineering Group (on the same office floor) that I have not mentioned so far for sharing part of the last few years (and in some cases, offices) with me: Sven Amann, Matthias Bahr, Richard Bubel, Ervina Cergani, Joscha Drechsler, Michael Eichberg, Matthias Eichholz, Leonid Glanz, Dominik Helm, Ben Hermann, Mirko Köhler, Florian Kübler, Patrik Müller, Radu Muschevici, Johannes Lerch, Jurgen van Ham, David Richter, Vanessa Rother, Jan Sinschek, Nathan Wasser, Pascal Weisenburger. Special thanks go to Sven Amann and Florian Kübler for managing STG's beverage supply, which included my crucial Club Mate supply.

Gudrun, the superhero among all the secretaries, deserves to be mentioned separately: Thank you for helping me come to STG, for running the group smoothly year after year despite all difficulties, for ensuring that no one ever has to worry about their contracts (I certainly never had to), and for your patience with everyone who repeatedly filled out the reimbursement forms for conference expenses wrongly. I wish you all the best for your well-earned pension years (not far away now)!

And then there are the MAIS people, without whom this long list could never be complete. They shared the very beginning of my PhD journey with me and taught me how to always get up again: Markus Aderhold, Kevin Bouhsard, André Fischer, Jinwei Hu, Steffen Lortz, Alexander Lux, Matthias Perner, Miriam Rifai-Schön, Jens Sauer, David Schneider, Dieter Schuster, Artem Starostin, Henning Sudbrock, Alexandra Weber, Sarah Werner - and, most of all, Heiko Mantel. Thank you for bringing me into research and for making me a fighter.

Finally, I thank my family and friends: My parents and my grand-parents for supporting me along the way and my long-time friend Laura Breitenstein for never leaving my side since school and for always being there for me during the hard times, even when my free time was very rare. My deepest love and thanks go to my husband Richard, whom I came to meet and know along the way to finishing this dissertation. Marrying you and starting a family with you clearly was the best outcome of all of this time and fully justified going through any crises and setbacks I experienced in between. Thank you for loving me unconditionally, for always believing that I would really finish this dissertation even at times when I did not believe it myself, and for our wonderful daughter Ilona, who, without knowing it, played a crucial role in helping me wrap up the thesis in the end. I love you both and I am very excited to tackle any future challenges with you!

Contents

Chapter

1

Introduction

In this thesis, we investigate the automated mechanization of soundness proofs for type systems of domain-specific programming languages (DSLs).

DSLs are nowadays routinely developed in practice [Fow11; Erd+13; EFO14], as productive programming languages for very specific application domains. Such domains are for example database queries and database manipulation (e.g. SQL), hardware description (e.g. Verilog), and web page design (e.g. HTML). A DSL allows a developer to express domain-specific concepts using only the terminology relevant for the domain in question and to ignore low-level technical details. Some DSLs, such as the ones we just mentioned, are widely known and used. Hence, they benefit from large developer teams or standardization committees as well as user feedback to improve the language's infrastructure (compiler, integrated development environments (IDEs), etc.) over time. Many other DSLs, however, are only developed and used within very specific contexts or companies. Such "small" DSLs are much more likely to suffer from severe correctness bugs than languages with large user bases.

Type systems are part of compilers for statically typed programming languages. The purpose of a type system is to detect a certain kind of bugs in programs, so-called "type errors", at compile time, i.e., prior to the execution of a program. Furthermore, type systems enable the implementation of important compiler optimizations, support the documentation of code, and play an important role for many IDE plugins that support developers (code completion, code recommendation, code-analysis tools, etc.).

Developers expect that if a type system does *not* report a type error for a program, no issues related to static types will appear during the execution of that program. Especially, the execution of a program shall not get stuck because the runtime environment encounters an ill-typed expression. For example, for most runtime environments, an expression such as `‘‘Hello’’ + 42` (numerical addition of a string and an integer) would lead to undefined behavior. So we expect that a type system reports an error when encountering this expression in the source code, preventing

its execution. We say that a type system that meets this expectation in the general case is *sound* [Pie02].

There are two main approaches to reliably ensure the soundness of a type system: The first approach is testing, with a high test coverage to cover as many cases as possible. This approach is adopted by many general-purpose languages such as Java. While testing works reasonably well in practice for languages developed by large teams and with large user bases, it is very likely to be incomplete for languages developed by small teams and with smaller user bases. Also, in theory testing can never truly ensure the absence of soundness bugs, since it is impossible to test all possible values of all variables in a program.

The second approach to reliably ensure the soundness of a type system is conducting a formal type soundness proof: We formally specify a type system using logical rules, formally specify type soundness as a logical property, and then logically prove that the specification indeed satisfies a soundness property. A canonical way to prove type soundness is via proving two syntactic properties called progress and preservation, the latter sometimes also called subject reduction [WF94; Pie02].

In a *mechanized proof*, a theorem prover checks every step against the basic rules of logical deduction, thus preventing common problems in human reasoning and raising the overall trust in the reliability of the proof. And naturally, the mechanization of proofs opens up potential for *proof automation*, ultimately reducing the overall effort to obtain a full proof.

Being able to automatically generate type soundness proofs is particularly interesting for the "small" DSLs we mentioned above: The proofs guarantee that the type systems within the DSLs function as desired, even without a high test coverage. The automation of type soundness proofs reduces the overall effort to obtain this guarantee, also for developers who do not have advanced skills in mechanized verification. However, the automation of such proofs is a very difficult open research problem. Hence, to the best of our knowledge, there is no solution for the automation of type soundness proofs, neither for DSLs, nor for general-purpose programming language today.

Our main, concrete goal in this thesis is a solution for obtaining a machine-checked soundness proof of a type system for a DSL with the least effort that is currently possible. Furthermore, we strive for obtaining a solution that may be reused and/or adapted for the automation of other verification domains, e.g. proofs of properties about program termination or security. Hence, the overall solution that we will present in this thesis targets what we will call *domain-specific verification* throughout this thesis. We call type soundness proofs of DSLs our *target verification domain*.

In the remainder of this chapter, we first introduce the general context of this thesis: We start with research context regarding the mechanization of type soundness proofs in general (Section 1.1). Next, we informally introduce our target verification domain further and introduce the general context of DSLs (Section 1.2). In Section 1.3, we introduce some first informal requirements for a solution that is suitable for automating different verification domains. Finally, we give an overview of the goals (Section 1.4), contributions (Section 1.5), and structure of this thesis (Section 1.6).

1.1. Mechanized Type Soundness Proofs

Mechanizing metatheory in programming languages has been an active research area since at least the 1980's, when verifiers such as NuPRL [Con+86] and Elf [Pfe91] (later Twelf [PS99]) were developed. We briefly summarize main research efforts in the area of mechanizing type soundness proofs, focusing on the overall effort that was needed for mechanizing individual proofs.

In most of the examples that we are going to mention, measuring the overall effort to obtain a mechanized proof was not the original goal, hence it is difficult to judge this effort in retrospect. As an approximation for measuring the effort, we will use the overall length of the final, mechanized proofs. Since the examples we present in this section were all developed with relatively little proof automation, this constitutes a good approximation of the overall human effort that was required.

1.1.1. Type Soundness Proofs of General-Purpose Languages

Research projects in which soundness proofs of type systems for general-purpose programming languages were fully mechanized typically reduce the languages in question to certain core features or even to minimal core calculi. Even so, the mechanizations tend to be rather long and complex, and typically involves a number of skilled researchers and students who invest months or even years of work time. We will give a number of examples to substantiate this claim throughout this section.

Around the year 2000, there was a lot of research on proving the type soundness of Java. Since Java itself is a large language with many complex features, typically only core features of Java were considered during mechanization. For example, advanced features like reflection and generics are often omitted (in past as well as in present research). A nowadays very famous minimal core calculus of essential Java features is Featherweight Java [IPW01a]. In 2006, Foster and Vytiniotis published a mechanization of the type soundness proof of Featherweight Java in Isabelle/HOL within the Isabelle *Archive of Formal Proofs* (AFP) [FV06]. Even this minimal formalization and the proof is about 46 pages long (including some minimal documentation) when printed on paper.

A larger subset of Java for which the type soundness proof was formalized at about the same time in Isabelle/HOL (in 2005) was Jinja [KN05]. The work on Jinja notably included a formalization of a type soundness proof for the JVM and of a compiler from Jinja to the JVM, thus capturing lots of realistic features of the Java language. This mechanization is substantially larger than the first one: The "shortened" version of the proof documentation in the AFP already comprises 157 standard pages when printed on paper. The full proof contains more than 20,000 lines of Isabelle code and was done by renowned Isabelle experts (Gerwin Klein and Tobias Nipkow) [KN06].

Another example of the large-scale mechanization of a type soundness proof was the mechanization of the metatheory of Standard ML [LCH06] in Twelf [PS99] from 2006. This mechanization comprises in total over 60,000 lines of Twelf code, developed by three Twelf experts.

1.1.2. The PoplMark Challenge

In the year 2005, leading researchers in the area of programming languages published the "PoplMark challenge" [Ayd+05], a benchmark for formalizing type soundness proofs for variants of *System F*, which is essentially a core calculus for any general-purpose programming language. Their goal was to make mechanized metatheory more accessible to programming language researchers at large, not only to researchers who are main experts within a certain verification system. The PoplMark challenge led to quite a number of successfully mechanized type soundness proofs in different systems and using different specification and proving strategies. Over the years, the work on the PoplMark challenge contributed to advancing the state of the art in interactive theorem proving, especially with regard to mechanizing meta theory in programming languages.

Concretely, the PoplMark challenge has three parts: The first two parts focus on mechanizing proofs. Of these two, the first part focuses on mechanizing proofs of properties of a specific type system feature, namely subtyping. The second part focuses on mechanizing the actual type soundness proofs (via progress and preservation). The third part specifies some meta requirements on what one should be able to use the mechanized metatheory for, such as code generation.

Solutions to the PoplMark *challenge* There exist different solutions in 6 verification systems. Most of them addressed only parts of challenge 1 and 2 and focused on proof of concepts rather than on generating complete solutions. The static webpage of the PoplMark challenge[1] summarizes all submitted solutions (not all of which were published). We discuss some details of the submitted solutions and the conceptual differences between them in Chapter 3 and only focus on the size of selected solutions for now.

Most solutions were submitted in Coq [Tea19]. The two most complete ones (addressing both of the first two parts of PoplMark) are the solution by Vouillon [Vou12] and the solution by Leroy [Ler07]. Vouillon's solution explicitly uses very little of the automated tactics available in Coq, focusing on the readability of the proofs. This makes the proofs of course longer than they strictly have to be in Coq, but explicitly shows the full creative effort needed for such a proof. Vouillon was mostly able to reproduce the proof structures given in the original PoplMark challenge, but of course had to be much more explicit for some parts of the proof, needing more lemmas. Overall, his proof comprises more than 4000 lines of Coq code. Leroy addresses only the simple parts of the PoplMark challenges 1 and 2, that is, fewer parts than Vouillon, but his proof still comprises about the same number of lines of Coq code as Vouillon's submission (about 4300 lines, using slightly more comments than Vouillon). Interestingly, Leroy notes in a comment about his solution, which can be found online[2]:

> "One bad thing about my solution: My Coq proof scripts do not have

[1] `https://www.seas.upenn.edu/~plclub/poplmark/`
[2] `https://www.seas.upenn.edu/~plclub/poplmark/leroy.html`

> the conciseness and elegance of Jérôme Vouillon's. Sorry, I've been using Coq for only 6 years..."

This citation shows that even Coq experts took a substantial time for devising their solutions to the PoplMark challenge, and then at least one of them still doubted the overall quality of his solution.

Next, there is a complete solution of PoplMark challenges 1 and 2 in Twelf [PS99] developed by three researchers (Michael Ashley-Rollman, Karl Crary, and Robert Harper), to be found online at the PoplMark webpage[3]. This solution is comprised of about 6700 lines of Twelf code, most of which are devoted to the more complex half of challenge 2 (over 4000 lines), the rest being relatively concise. In a comment on their solution (also online, see before), the authors note:

> "The Twelf methodology (worlds, world subsumption, termination checking, etc.) can be a bit mysterious for the uninitiated - not only for writing proofs, but even for understanding the significance of what one is reading."

So again, also for developing and understanding Twelf proofs, special skills are essential, even from the point of view of "system insiders".

Stefan Berghofer devised a full solution to all three parts of the PoplMark in Isabelle/HOL [Isa18], available in the Isabelle AFP [Ber07]. It comprises about 4600 lines of Isabelle code and hence is relatively concise for being a full solution. Still, also Berghofer is an Isabelle expert and had the advantage of advanced knowledge of various relevant Isabelle internals (e.g. code generation, which was relevant for addressing part 3 of PoplMark etc.).

We note that proof automation was not the main focus of the PoplMark challenge. Hence, all of the solutions whose size we sketched above were developed interactively by humans within the respective verification systems. Each proof mechanization was an entire research project of its own. Some aspects of the proofs for the PoplMark challenge were automated in separate research projects, but no full automation was ever achieved. To the best of our knowledge, there have been no further officially submitted solutions to the PoplMark challenge and also no further centralized attempts to achieving the full automation of type soundness proofs since 2012.

Our intermediate conclusion from studying the PoplMark challenge and its solutions is: Mechanizing soundness proofs of type systems for general-purpose languages with existing verification systems is very hard and requires a lot of effort and skill - even if one only mechanizes core aspects of a general-purpose language. This was true a decade ago, but has not changed in its essence until today. Then and today the hope that proofs such as the ones mentioned here could ever be fully automated seems highly unrealistic.

In Chapter 3, we analyze the state-of-the-art in mechanized type soundness proofs in more detail and break down our main goal of automated type soundness

[3] `https://www.seas.upenn.edu/~plclub/poplmark/cmu.html`

proofs for DSLs: We identify and explain the main problems that occur during the mechanization of general type soundness proofs (in short, name binding). We use our observations to define a relevant subset of DSLs for which the automated mechanization of type soundness proofs is feasible (in short, languages without first-class binders). Furthermore, we mechanize a type soundness proof of an example DSL ourselves within two existing verification systems and investigate the effort needed for this mechanization as well as the degree of automation that these systems allow for our example proof. The results from the survey in Chapter 3 serve as the base for the main contributions of this thesis.

1.2. Domain-specific Languages (DSLs)

As mentioned previously, this thesis targets the automation of type soundness proofs of DSLs, especially of DSLs developed by small teams and with small user bases. Such "small" DSLs would benefit enormously from a solution for automatically generating type soundness proofs for improving the overall quality of language environments. In this section, we motivate and introduce our target verification domain further.

1.2.1. Definition, Usage, and Types of DSLs

A *domain-specific programming language* (DSL) explicitly offers language constructs designed for specific application domains, abstracting away from low-level general-purpose details. For example, a DSL for querying and manipulating databases such as SQL offers specific syntax for row selection and column projection. A DSL for hardware description such as Verilog offers specific syntax for hardware primitives. A DSL for web page development such as HTML offers specific syntax for describing visual elements on web pages. The compiler and/or runtime environment of a DSL implements the low-level technical details of such domain-specific constructs. Developers who want to implement an individual task within a certain domain may choose an appropriate DSL and use its syntax without coming into contact with technical details that are unrelated to the respective domain.

In contrast, *general-purpose programming languages* such as C, Haskell, Java, Scala etc. are designed for developing a wide range of applications. Hence, such languages offer a core set of general computation and branching operations. One may use these low-level operations to implement complex, domain-specific tasks such as graphical user output, database manipulation and queries, etc. Developers can combine such individual tasks within their own programs, using the syntax of the respective general-purpose language. Thus, developers who want to implement individual tasks within a specific application domain need to be able to use the host general-purpose language as well.

DSLs are especially useful in working contexts with many experts for a certain domain which lack universal technical programming skills. For these experts, it is easier to understand code and learn a DSL with only the specific concepts necessary for their domain than to learn a fully-fledged general-purpose language.

But also universal programming experts may profit from domain-specific language abstractions for raising their productivity. A DSL may also be useful to bridge the knowledge gap between these two groups of experts. Some DSLs, such as the ones mentioned above, are widely used in different companies. Other DSLs are developed and used only within specific companies. For example, many insurance companies have internal DSLs for describing different insurance policies.

There are two main types of DSLs: A DSL can either be an *external* or *standalone language*, with its own compiler and/or runtime environment and its own development infrastructure. Examples of external DSLs are SQL and HTML. Or a DSL may be implemented as an *internal* language, also called an *embedded* DSL. In the latter case, the language constructs of the DSL are embedded into a host language, often a general-purpose language, using the runtime environment and the development infrastructure of the host language. Embedded DSLs may easily be integrated with other infrastructure or other embedded DSLs via their host language. On the other hand, using embedded DSLs often requires developers to also be acquainted with the host language to a certain degree.

The distinction between a DSL and a general-purpose language is not always clear. For example, any *API* (application programming interface) can be seen as an embedded DSL. An API is typically implemented within a general-purpose language and serves as a developer interface to a special-purpose library or application. The methods within an API allow for accessing specific functionality from such libraries or applications while abstracting from the technical implementation details of this functionality.

An extensive reference on the overall topic of DSLs and different types of DSLs is Fowler's book [Fow11].

In this thesis, DSLs are going to appear in two different contexts: On the one hand, we automate type soundness proofs for a certain kind of DSLs. In this context, we will focus on *external* DSLs. On the other hand, we also make use of DSLs within the implementation of our final solution for the automation of our target verification domain. In this context, we use *embedded* DSLs.

1.2.2. Type soundness proofs of DSLs

In practice, many DSLs are not equipped with a type system. For example, SQL queries are not typed and hence may fail at runtime if they refer, for instance, to a non-existing column in a table. Yet, the addition of a type system improves the general usability of such DSLs: A type system may help new users to learn a DSL more quickly, and may help more experienced users to avoid careless mistakes while coding. The present type information typically helps with understanding and maintaining the resulting code. These claims are backed by numerous scientific empirical user studies that compare different usability aspects of languages with and without static typing against each other, e.g., the work of Hanenberg et al. [Kle+12b; Han+14; End+14].

However, as we argued already at the beginning of this chapter, a type system can only be truly useful if developers also invest resources into ensuring its soundness -

either via extensive testing, or, more reliably, by developing a mechanized soundness proof. The latter is beyond the skills of the typical DSL developer, and both is often beyond the available resources of the developers. This is even more true for the development of small DSLs.

If we informally consider the type systems that would be needed for many DSLs, especially small ones, we find that they would be conceptually simple. For example, a type system for SQL queries would simply have to check how individual query operations will manipulate the schemas of tables (e.g. drop and add table columns) and make sure that these manipulations align well with other query operations present. The conceptual simplicity of the required type systems is a direct consequence of the conceptual simplicity of many DSLs: These languages are supposed to restrict themselves to the core concepts relevant for their domain, avoiding complicated programming language features such as generics, object orientation etc. that complicate the corresponding type systems.

So it is not surprising that type soundness proofs for such conceptually simple type systems are also conceptually straightforward, following a standard scheme. The main effort in developing such soundness proofs lies in making sure that all cases of the proof are covered. No creative reasoning techniques are required: Standard induction and case distinction techniques together with the application of auxiliary lemmas that follow a specific scheme suffice. One contribution of this thesis is to describe this standard scheme. We will see this description and concrete examples in Chapter 6 and in Chapter 7.

This overall situation is precisely what makes the target verification domain of this thesis attractive for automation: On the one hand, it is worthwhile to provide a solution that simplifies the development of soundness proofs of type systems for DSLs. An automated solution may raise the overall number of sound type systems for DSLs and improve the usability of these languages. On the other hand, the conceptual simplicity of type soundness proofs for DSLs makes their full automation feasible, at least for some DSLs. In Chapter 3, we will describe the subset of DSLs that we consider in this thesis further.

1.3. Domain-specific verification

As mentioned at the beginning of this chapter, we are not only interested in arriving at a solution for automating our target verification domain, but also in arriving a solution that can in principle be reused for automating other verification domains. In short, we are interested in a generic solution useful for automating *domain-specific verification* tasks. In this section, we will outline first informal requirements for such a generic solution for domain-specific verification and briefly put them into contrast with the current state of the art in verification. Chapter 3 will go into more details regarding current verification systems, while Chapter 4 will go into more details regarding the requirements for a generic solution for automating domain-specific verification tasks.

1.3.1. Verification Infrastructure Instead of Standalone System

When discussing the PoplMark challenge above, we already briefly mentioned a number of existing *standalone* interactive theorem provers and verification systems/platforms. These systems are in general installed as *separate*, independent systems. Users have to specify their input specifications or programs in the format provided by the system, prove properties on their specifications within the system, and finally obtain proofs which are only directly reproducible within the verification system used. Thus, the verified specification or program becomes an artefact that only "lives" within the verification system used and often has no direct connection to the program or system used in the real world. (Some systems such as Isabelle and Coq provide indirect connections by offering the possibility of generating code in other languages from specifications.)

In contrast to a standalone verification system, we are looking for a *verification infrastructure* which may easily be integrated with other systems. Ideally, this verification infrastructure should be a lightweight library in a versatile and known programming language. Such a verification infrastructure may easily be extended and improved to accommodate the needs of different verification domains.

1.3.2. Target Groups: "Domain Experts" and "End Users"

The generic solution we are looking for should accommodate two target groups. Our primary target group is what we call a "domain expert": A *domain expert* is a researcher or developer who plans to automate a certain kind of proofs. We assume that a domain expert has seen or done a lot of proofs in a certain domain and hence has a reasonable idea on the general steps and techniques necessary for proofs in that domain. The motivation of domain experts to automate a certain kind of proofs may either be to facilitate further verification projects of their own, or to facilitate verification projects of "non-experts" in the given domain. These "non-experts" constitute our secondary target group, which we will call "end users" in this thesis. End users need not have any special verification skills for the domain, but just want to apply automated strategies to verify properties in a certain verification domain.

Our solution for automating domain-specific verification tasks shall primarily target domain experts, but have end users in mind as well.

We assume that domain experts for the target verification domain we consider in this thesis have a strong background in programming languages research or development. In particular, we are going to assume that domain experts from this area

- know one or more general-purpose programming languages quite well, notably functional programming languages,
- know several DSLs,
- have seen or developed themselves several specifications of type systems for DSLs using inference rule notation,

- have seen or developed themselves soundness proofs of such type systems, via progress and preservation properties, and
- have some basic knowledge in using theorem provers and other verification tools for formalizing such proofs.

We emphasize that we *do not* assume that domain experts in a particular verification domain have expertise in using advanced features such as tactic languages for complex automation tasks. Also, we *do not* assume that domain experts are familiar with the internals of existing theorem provers, i.e., with their implementation.

The assumptions that we outlined for domain experts correspond to the expertise of various researchers that we talked to during the last few years. According to our observations, there is a significant gap in knowledge and skills between, on the one hand, domain experts in different domains, and experts in mechanized theorem proving on the other hand. The main reason for this is probably that both groups tend to work within their own communities, with their own conferences and research groups. Exchange between these groups exists, but not on a large scale. We believe that it is due to this situation that a large number of well-understood verification domains, such as our target verification domain in this thesis, has not been automated to date.

Our generic solution for automating domain-specific verification tasks shall bridge the gap between domain experts and experts in existing verification systems.

1.3.3. Different Domains – Different Input Formats

As a first step toward bridging the gap between domain experts and experts in existing verification systems, our solution shall enable domain experts to employ their own format for specifying input problems. Such a format can be an input DSL for problem specifications in a particular domain. For instance, for specifying a type system, one may like to use an input format that offers a specific notation for inference rules and specific syntax for typing judgments used in the literature. For other verification domains, other domain-specific syntax constructs are useful for domain experts.

Furthermore, an input format should in principle enable the addition of proof-relevant information to a problem specification. For example, an input format may allow end users to annotate their specifications with hints for the verification automation. The possibility to use such hints may raise the number of individual problems that a domain-specific automated strategy for verification can cover. In different verification domains, domain experts as well as end users may find different syntax intuitive for giving such hints for the verification automation.

Current verification systems target general-purpose verification. They typically offer a single general-purpose specification format for specifying input problems. This specification format sometimes allows for including some general hints for verification strategies, such as for example commands that indicate which definitions and lemmas an automated simplifier should include and try by default and which not. The design of existing systems does not account for extending the format for

input specifications with custom language constructs or specification annotations. Additionally, since these systems target any numbers of different verification domains, they tend to be rather large and complex. For example, systems such as Isabelle and Coq possess a complex, verified core whose structure and rules need to be known and respected by anyone who extends the system. This complexity makes it difficult for domain experts to attach and integrate their own domain-specific format.

Our solution for automating domain-specific verification tasks shall explicitly allow domain experts to flexibly develop and integrate their own domain-specific formats for input problems.

1.3.4. Languages and Methods for Verification Automation

As second step toward bridging the gap between domain experts and experts in existing verification systems, it is important to give domain experts some flexibility regarding which language and methods they can use for implementing strategies for automating their verification domain. Ideally, they should be able to use a language that they know already. For many domains, including our target verification domain, this will likely simply be a wide-spread general-purpose programming language, such as Java or Scala.

For other domains, it may even be useful to enable domain experts to provide a DSL specifically for automating verification tasks in their domain. Such a DSL may include specialized syntax for verification techniques that are common within that domain. Domain-specific verification techniques may for example be the application of certain induction schemes, or the application of domain-specific reasoning techniques such as the unwinding technique (used for instance in proofs in the area of information-flow security) or logical relations (used for instance in proofs of certain properties of programs in a specific language such as strong normalization). Domain-specific verification techniques may also include the application of certain templates for generating common auxiliary lemmas in a verification domain. A specific DSL for strategies in a particular verification domain may include syntax primitives for special proof techniques as well as for describing templates for specific proof steps or auxiliary lemmas.

Method-wise, different abstract techniques may be intuitive for domain experts in different verification domains. For some verification domains, it may be most intuitive for the corresponding domain experts to implement automated strategies that generate a proof “bottom-up”, e.g., by first generating a number of overall laws and lemmas and then attempting to combine these in a clever way for solving an overall problem. For other domains, it may be most intuitive to generate a proof “top-down”, i.e. by starting from a top-level problem and iteratively decomposing it into smaller problems. For some domains, it may be most intuitive to implement automated strategies as a sequence of “methods” to apply for transforming an overall goal. For other domains, it may be most intuitive to generate the concrete intermediate *steps* that should occur within a proof.

Existing interactive theorem provers offer tactic languages for automating general-purpose verification tasks. However, using a tactic language usually requires a

deeper technical understanding about the internals of the corresponding interactive theorem prover. Tactic languages do not provide the means to develop DSLs for verification that truly abstract from domain-specific concepts. To some degree, a domain expert with a deeper knowledge of a prover's internals may of course use its tactic language to develop a tactic API for a certain verification domain. However, if another domain expert or also an end user needs to touch one of the tactics because a special case is not working, this person invariably has to look deeper into the technical details, requiring some knowledge of the prover's internals. Hence, the existing tools and platforms neither provide a low entry point for domain experts for developing DSLs for verification in the first place, nor do they enable end users to truly forget about the underlying technical general-purpose aspects of proving.

Finally, tactic languages enforce a certain method for implementing a verification strategy - in short, top-down transformation of top-level goals by describing a series of methods for transforming the goals. Thus, tactic languages provide little methodological flexibility to domain experts.

Our solution for automating domain-specific verification tasks shall offer domain experts as much flexibility as possible regarding which language and methods they can use for implementing automated strategies for verifying properties in their domain. Ideally, it should offer the possibility of adding and using a custom DSL for implementing such automated strategies in a specific verification domain.

1.4. Goals of This Thesis

Figure 1.1 gives an overview of this thesis via a matrix-like structure. On the vertical axis, we have the three meta stages of this thesis that build on each other. First, on top, comes the definition of the overall goals of this thesis. Next, in the middle, comes how we break the overall goals down into individual research problems. Finally, on the bottom, we have the final contributions of this thesis. On the horizontal axis, we have the level of "concreteness" of the individual parts of this thesis: Parts more to the left are more abstract, while parts more to the right are more concrete, i.e., talking about concrete examples. Some of the individual boxes within this matrix contain pointers to the chapters within this thesis that correspond to a specific part. We will refer to Figure 1.1 in the remainder of the introduction to give an overview about the goals, contributions, and structure of this thesis.

Our concrete goal in this thesis is the automation of type soundness proofs via progress and preservation of DSLs. We refer to this particular verification domain as our *target verification domain* throughout this thesis. We divide this overall goal into a number of concrete sub-goals, spelt out in the box on the right-hand side of the "Problem" part in the middle of Figure 1.1: Firstly, we aim at discovering and describing a subset of DSLs for which it is feasible to achieve a high degree of automation for type soundness proofs, since full automation of such proofs for arbitrary languages is undecidable in general. Next, we aim at describing the overall structure of type soundness proofs for such DSLs and how such proofs may be

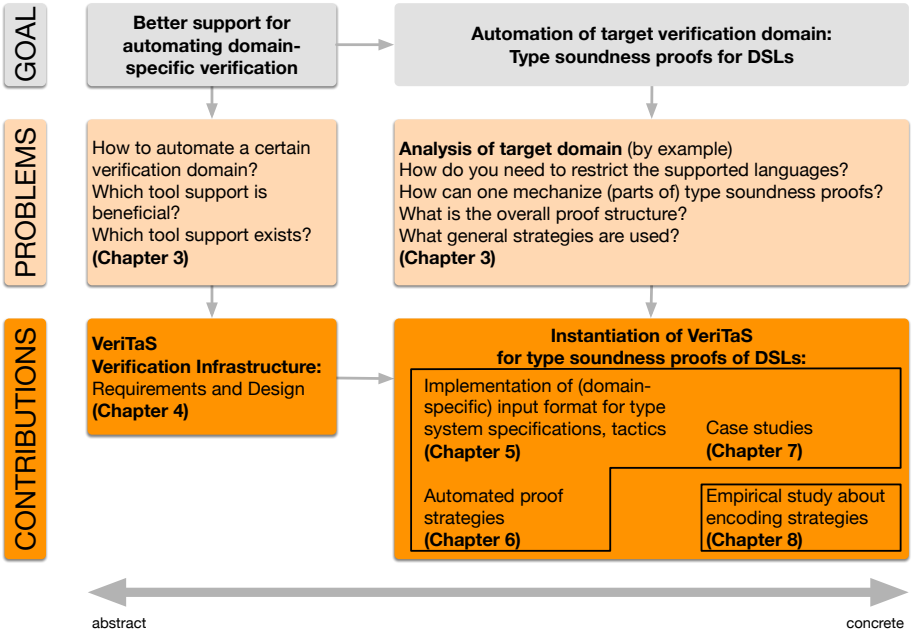


Figure 1.1.: Overview of this thesis

mechanized.

Our abstract, secondary goal in this thesis is to improve the overall tool support for automating domain-specific verification tasks. Or, as we put it earlier in this chapter, to arrive at a solution for automating our target verification domain that may be reused for automating other verification domains as well. For this abstract goal, our sub-goals are to formulate requirements for a solution that supports the automation of different verification domains by domain experts and to study which existing tool support is beneficial for arriving at such a solution.

1.5. Overview of Contributions

The middle and the lower part of Figure 1.1 summarizes the contributions of this thesis. Firstly, this thesis contains abstract as well as concrete contributions that break down the overall goal of automated type soundness proofs into sub-problems, defining sub-goals and steps to take toward the main automation goal ("Problems" part): We contribute the definition of a subset of DSLs for which the automation of type soundness proofs is feasible and describe the structure of the mechanized type soundness proofs for such DSLs. Furthermore, we contribute an analysis of existing verification systems and how well they are suited for the automation of domain-specific verification tasks by domain experts who do not necessarily have insider knowledge of the respective systems. We concretize our definitions and analyses by conducting a full example progress and preservation proof of a typed

subset of SQL within two existing verification systems.

Based on our problem analysis, we contribute the requirements, design, and prototypical implementation of a generic verification infrastructure called VeriTaS, that is designed for supporting the development of automated proof strategies for different verification domains. VeriTaS is a library implemented in Scala that enables automatically breaking down domain-specific proofs into different smaller proof problems and to structure all of these problems within intuitive, human-readable proof graphs. The infrastructure allows for interacting with different existing provers. It is generic in a domain-specific input format for problem specifications.

Next, we contribute a concrete instantiation of our VeriTaS verification infrastructure. The individual elements of this instantiation focus on our target verification domain, but may also be reused and adapted for other verification domains. Our instantiation consists of a domain-specific input format for type-system specifications, reusable tactics for basic steps within type soundness proofs, and encoding strategies for low-level proof steps to input formats supported by existing automated theorem provers. Secondly, it consists of proof strategies that automatically construct a high-level proof structure within VeriTaS for progress and preservation proofs for our target languages. Our proof strategies expect a specification of a DSL's reduction semantics and type system, domain-specific user annotations, and appropriate auxiliary lemmas. We contribute two different case studies which demonstrate that our strategies indeed automatically generate large parts of progress and preservation proofs.

Finally, we contribute a thorough analysis of a part that turned out to be crucial for the overall functioning of our automation approach: the encoding of low-level proof steps to the input format of existing automated theorem provers. To this end, we conducted an empirical study to compare different encoding strategies for different categories of simple test problems on specifications of type systems of DSLs. Our study empirically confirms that the choice of an encoding strategy is crucial to the overall automation success and provides guidance as to which strategies are beneficial for problems from the target verification domain of this thesis.

1.6. Structure of This Thesis

In Chapter 2, we introduce the relevant concepts needed to understand the contributions of this thesis and their description: the formal specification of type systems and type soundness via progress and preservation, an introduction to Scala (the programming language used for implementing VeriTaS), and automated as well as interactive theorem provers.

Figure 1.1 also indicates the overall structure of the content chapters within this thesis: In Chapter 3, we present a survey on the state of the art for mechanizing type soundness proofs and restrict the set of DSLs we consider in this thesis.

Chapter 4 presents the requirements and design of VeriTaS, our generic verification infrastructure for enabling the development of automated domain-specific proof strategies for different verification domains.

In Chapter 5, we present a concrete instantiation of our verification infrastructure for the target verification domain of this thesis. Notably, we connect existing automated provers to VeriTaS for verifying low-level steps in proofs. We take care that the elements of this instantiation may be reused for other verification domains. Chapter 6 presents the implementation of automated proof strategies within our instantiation of VeriTaS. Our strategies automatically generate high-level proof structures. Again, we focus on developing automated proof strategies for our target verification domain, but take care that core elements of our implementation for these strategies may be reused for other verification domains. Chapter 7 presents and analyzes our case studies for the automated strategies from Chapter 6, using concrete examples from our target verification domain.

Chapter 8 presents our empirical study of encoding strategies for low-level proof steps into first-order logic. Chapter 9 discusses work related to what we present in this thesis and Chapter 10 concludes with an overall summary and a discussion of possible directions for future research on the topics treated within this thesis.

Chapter

2

Background

We briefly outline the scientific context of programming language (PL) research in which this thesis was developed and introduce relevant concepts from this context: We start with introducing first-order logic (FOL), the logic into which we translate low-level proof problems within this thesis and which also constitutes the notational basis for language specifications (Section 2.1). Next, we explain type systems (Section 2.2) and soundness proofs of type systems (Section 2.3), the target verification domain within this thesis. Then we introduce the key syntax of Scala, the programming language that we used for implementing the concepts presented in this thesis (Section 2.4). Finally, we present basic concepts from automated theorem proving (Section 2.5) and interactive theorem proving (Section 2.6).

2.1. First-order Logic (FOL)

In this thesis, we will encode proof problems into different variants of *classical* first-order logic. First-order logic also constitutes the basic notation from which the notation for language specifications that we introduce in the following subsection is derived. We summarize the notations, their informal definition, and the notational abbreviations that we use in the following chapters. A fully formal introduction to first-order logic can be found in the book by Ebbinghaus et al. [EFT94].

The syntax of *untyped first-order logic* consists of terms and formulas. A *term* consists of constant symbols, n-ary function symbols, and variable symbols. A *formula* is inductively defined on terms and n-ary relational symbols: A formula is

- an n-ary relational symbol,
- the equality (denoted by $t_1 = t_2$) or the inequality (denoted by $t_1 \neq t_2$) of two terms t_1 and t_2,
- logical *negation* $\neg\phi$ of a formula ϕ,

- logical *conjunction* $\phi \wedge \psi$ of two formulas ϕ and ψ,
- logical *disjunction* $\phi \vee \psi$ of two formulas ϕ and ψ,
- logical *implication* $\phi \implies \psi$ of two formulas ϕ and ψ (which abbreviates the formula $\neg\phi \vee \psi$),
- logical *bi-implication* $\phi \iff \psi$ of two formulas ϕ and ψ (which abbreviates the formula $\phi \implies \psi \wedge \psi \implies \phi$),
- *universal quantification* $\forall v.\ \phi$ for expressing that a formula ϕ holds for all possible interpretations of a variable v, and
- *existential quantification* $\exists v.\ \phi$ for expressing that a formula ϕ holds for at least one interpretation of a variable v.

In *typed* first-order logic, we add the signatures of all used constant, function, and relation symbols to a set of formulas. We specify a signature for an n-ary function symbol with the syntax $s_1 \times s_2 \times \ ... \ \times s_n \to s$ to denote that the function takes n arguments of the respective sort and returns a term of sort s. Furthermore, we add the sorts of variables in quantified formulas: $\forall v : S.\ \phi$ denotes that variable v has sort S.

As abbreviations, we omit variable sorts in quantified formulas even when working in typed first-order logic to keep the formulas shorter. Furthermore, we use the following abbreviations for longer formulas:

- $T_1 ... T_n \to T$ abbreviates a function signature of the shape $T_1 \times T_2 \times ... \times T_n \to T$
- $\forall t_1 : T_1, t_2 : T_2, ..., t_n . T_n.\ \phi$ abbreviates $\forall t_1 : T_1.\ \forall t_2 : T_2.\ ...\ \forall t_n : T_n.\ \phi$ (and similarly for existentially quantified formulas)
- $\overline{a}$ abbreviates the list $a_1, ..., a_n$ of variable or argument symbols; e.g. an n-ary function application $f(p_1, ..., p_n)$ may be written as $f(\overline{p})$, and an n-ary universal quantification $\forall v_1, ..., v_n.\ \phi$ as $\forall \overline{v}.\ \phi$ (and similarly for existential quantification)
- $\bigvee_i \phi_i$ or $\bigvee_{i \in \{1..n\}} \phi_i$ for $\phi_1 \vee ... \vee \phi_n$
- $\bigwedge_i \phi_i$ or $\bigwedge_{i \in \{1..n\}} \phi_i$ for $\phi_1 \wedge ... \wedge \phi_n$

Note that in this thesis, we solely work in *classical* logic, i.e. we assume that the formula $\phi \vee \neg\phi$ is always true.

2.2. Type Systems

In practice, type systems are implemented within compilers of statically typed programming languages to check programs for certain so-called "type errors" at compile-time, i.e. before they are executed. Typical type systems check for example

```
t ::=                    v ::=
  true                     true
  false                    false
  if t then t else t       nv
  0
  succ t                 nv ::=
  pred t                   0
  iszero t                 succ nv
```

Figure 2.1.: Syntax of typed arithmetic expressions in standard notation from PL literature

that operations only receive arguments that they can handle, for example that a numeral addition operation only receives numbers. For these checks, the type system only inspects the syntax of a program. If it finds a type error within the syntax (for example the attempt to add a string to a number), it reports this error during compile time, preventing the execution of the erroneous program and thus the occurrence of potentially undefined behavior during a program run.

In this thesis, we are interested in logically proving properties about type systems (we will expand this point in the following section). For this, we require formal specifications of type systems. These in turn build on formal specifications of the syntax and of the reduction semantics of a programming language. We introduce these concepts by example with the notation used in Pierce's "Types and Programming Languages" (TAPL) [Pie02], a standard reference in the area of programming languages and type systems. As an example, we use Pierce's specification of *typed arithmetic expressions*, from Chapter 3 and 8 of TAPL. We will use this example as a running example throughout this thesis for explaining and demonstrating our contribution. We refer the interested reader to Pierce's book for more details on the topic of type systems.

2.2.1. Language syntax

In the literature on programming languages, a language's syntax is typically specified using grammars. We define the syntax of typed arithmetic expressions in Figure 2.1 via a set `t` that represents the inductively defined set of terms. The set contains simple boolean expressions (the values `true` and `false`) as well as a conditional expression. Furthermore, it contains constructs for building (natural) numbers (0 as base value, `succ` for incrementing and `pred` for decrementing a number). Finally, there is a construct for checking whether a term `t` is equal to zero. We also define the set `v` of values within the language (i.e. constructs that will not be reduced further), which contains the inductively defined set `nv` of *numerical* values.

Alternatively to using grammars for describing a language's syntax, we may also use *algebraic datatypes* (ADTs), i.e. datatypes defined via a set of constructors with zero or more arguments. Constructor arguments may again be of the type we are defining to form a recursive datatype, with which we can model inductively defined sets. For describing subsets of terms such as values and numerical values,

$$\frac{}{\texttt{if true then } t_2 \texttt{ else } t_3 \rightarrow t_2} \text{ (E-IfTrue)}$$

$$\frac{}{\texttt{if false then } t_2 \texttt{ else } t_3 \rightarrow t_3} \text{ (E-IfFalse)}$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \text{ (E-If)}$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{succ } t_1 \rightarrow \texttt{succ } t_1'} \text{ (E-Succ)}$$

$$\frac{}{\texttt{pred 0} \rightarrow \texttt{0}} \text{ (E-PredZero)}$$

$$\frac{}{\texttt{pred (succ } nv\texttt{)} \rightarrow nv} \text{ (E-PredSucc)}$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{pred } t_1 \rightarrow \texttt{pred } t_1'} \text{ (E-Pred)}$$

$$\frac{}{\texttt{iszero 0} \rightarrow \texttt{true}} \text{ (E-IsZeroZero)}$$

$$\frac{}{\texttt{iszero (succ } nv\texttt{)} \rightarrow \texttt{false}} \text{ (E-IsZeroSucc)}$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{iszero } t_1 \rightarrow \texttt{iszero } t_1'} \text{ (E-IsZero)}$$

Figure 2.2.: Reduction semantics of typed arithmetic expressions in standard notation from PL literature

we can then define predicates on the top-level algebraic datatype for terms. We will see numerous examples for such definitions throughout the thesis, for example in Section 5.1.1.

2.2.2. Reduction semantics

When defining the syntax of typed arithmetic expressions, we already hinted at the intended semantics of the different language constructs. We formalize these intuitions via a set of *inference rules* for a reduction relation $t \rightarrow t'$, which models that a term t reduces in one step to a term t'. Inference rules are a notational variant of logical implications: All premises above the line together (logical conjunction) imply the conclusion below the line. Furthermore, all variables within the inference rule are implicitly universally quantified. If there is no premise above the line, this abbreviates `true`, i.e. the conclusion of the inference rule is an axiom.

In Figure 2.2, we define what is called in the PL literature an "operational *small-*

$$\frac{}{\texttt{true} : \texttt{Bool}}\ (\text{T-True})$$

$$\frac{}{\texttt{false} : \texttt{Bool}}\ (\text{T-False})$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T}\ (\text{T-If})$$

$$\frac{}{0 : \texttt{Nat}}\ (\text{T-Nat})$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}}\ (\text{T-Succ})$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}}\ (\text{T-Pred})$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}}\ (\text{T-IsZero})$$

Figure 2.3.: Type system for typed arithmetic expressions in standard notation from PL literature

step reduction semantics": Operational, since we define syntactical steps on how to reduce each term in the form of rules that we could directly implement. And the semantics is in a small-step style since it describes individual steps of computation to rewrite a term step by step until it eventually becomes a value (that cannot be reduced further). As opposed to the small-step style, reduction semantics may also be defined as *big-step semantics*, which describe in a single rule how a term evaluates to a final value. We do not consider big-step semantics in this thesis.

The reduction semantics of a language is also called the *dynamic semantics* of a language in the literature, as opposed to its static semantics, which we introduce next.

2.2.3. Typing rules

We say that a term of a language is *stuck* if the term has been reduced as far as possible with the inference rules from the dynamic semantics, but is not a value. For example, the term `pred true` is a stuck term. We define a *type system* for a language in order to detect stuck terms purely from inspecting their syntax without actually having to reduce a term. For this, we first introduce the notion of type: A type *classifies* terms according to the class of values to which they would evaluate. In our example specification, we use two types, `Nat` for natural numbers and `Bool` for boolean values. We introduce a *typing judgment* $t : T$ to say that "a term t has type T".

Figure 2.3 defines the type system for typed arithmetic expressions by specifying

typing rules, also in the inference-rule notation. Note that type systems typically are *conservative*, i.e. they may not be able to type-check all terms that would reduce to a value without getting stuck. For example, the term `if (iszero 0) then 0 else false` will successfully reduce to `0`, but cannot be typed by the typing rules from Figure 2.3. The type system we define above is a so-called *syntax-directed* type system: There is exactly one typing rule for each syntactical language construct.

A type system is also called a language's *static semantics* in the PL literature. More complex type systems may use a typing judgment with more than two arguments, for example with a context Γ for storing the variable bindings available within the current scope. We will see examples of such typing judgments later in the thesis.

2.3. Type Soundness via Progress and Preservation

Implementations as well as formal specifications of type systems may contain errors just like implementations or specifications of any other program. However, ideally one wants to ensure that every program that a type system successfully checks against a type will really not reduce to a stuck term, or, differently said, that "well-typed terms do not go wrong". This idea is called *type soundness*, or also *type safety*. We may formally specify the soundness of a type system and then logically prove it. In the PL literature, type soundness is typically formulated in two steps, called progress and preservation, going back to the work of Wright and Felleisen [WF94].

- *Progress* intuitively states that a well-typed term that is not yet a value can always take another reduction step.
- *Preservation* intuitively states that if a well-typed term is reduced one step further, the result of this reduction is also well-typed (typically with the exact type the term had before the step, hence the name "preservation").

We now define progress and preservation formally for typed arithmetic expressions and sketch how one can prove them on paper.

Theorem 2.1. *(Progress)* $t : T \implies \mathit{isvalue}(t) \lor \exists t'.\ t \to t'$ $\Diamond$

Proof: The proof proceeds by induction on a derivation of $t : T$. Since the type system for typed arithmetic expressions is syntax-directed, this is equivalent to a structural induction over term t.

Cases T-True, T-False, T-Zero*:* Here it holds that $t = \texttt{false}$ or $t = \texttt{true}$ or $t = 0$. These terms are all values, hence progress trivially holds.

Cases T-If, T-Succ, T-Pred, T-IsZero*:* It holds that

- $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$ or

- $t =$ `succ` t_1 or
- $t =$ `pred` t_1 or
- $t =$ `iszero` t_1

In all of these cases, the induction hypothesis for t_1 gives us that t_1 is either a value or there is a t' so that $t \to t'$. Then, for T-If, we can either apply rule E-IfTrue or rule E-IfFalse if t_1 is a value (since t is well-typed, t_1 has to be either `true` or `false`). Or we may apply rule E-If if t_1 is not a value. Similarly, for T-Succ, we either already have a value (t_1 has to be a numerical value in that case), or we may apply rule E-Succ to take a step. For T-Pred, either rule E-PredZero or rule E-PredSucc lets us take another step if t_1 is a (numerical) value, or we may evaluate t further by applying E-Pred. Similarly for case T-IsZero.

Theorem 2.2. *(Preservation)* $t : T \ \wedge \ t \to t' \implies t' : T$ $\Diamond$

Proof: The proof proceeds by induction on a derivation of $t : T$, which here is equivalent to a structural induction on t.

Cases T-True, T-False, T-Zero*:* Here it holds that $t =$ `false` or $t =$ `true` or $t = 0$. These terms are all values, hence preservation trivially holds since these values cannot take another step.

Case T-If*:* It holds that $t =$ `if` t_1 `then` t_2 `else` t_3. So t can only take a step via the rules E-IfTrue, E-IfFalse, or E-If. We consider these sub-cases separately:

- Sub-cases E-IfTrue, E-IfFalse: Term t reduces either to t_2 or to t_3. Both of these terms also have type T, according to the premises of typing rule T-If, so the conclusion holds.
- Sub-case E-If The result of the reduction step is $t' =$ `if` t_1' `then` t_2 `else` t_3. From the induction hypothesis for t_1, we get that the term t_1' to which t_1 reduces also has to have type `Bool`, like t_1. So we may type t' with T again and the conclusion holds.

Cases T-Succ, T-Pred, T-IsZero*:* Either the reduction step takes place by one of the rules with no premises, i.e. E-PredZero, E-PredSucc, E-IsZeroZero, or E-IsZeroSucc. All of these rules evaluate the given term to a term with the same type as before. Or the reduction step takes place via one of the rules with premise $t_1 \to t_1'$ (i.e. E-Succ, E-Pred, or EZero). In these cases, we apply the respective induction hypothesis for t_1., which gives us that any term to which t_1 reduces has the same type as t_1. This allows us to type the overall result of the step of t again with the type of t (like in sub-case E-If).

2.4. Scala

In this thesis, we use Scala to implement our verification infrastructure. *Scala* is a general-purpose programming language that runs on the Java virtual machine and is compatible with the Java infrastructure (libraries and also IDEs). Scala is a functional, object-oriented language. The canonical reference for the Scala language is the book on Scala by Odersky et al. [OSV11].

Here we briefly introduce the basic language constructs of Scala which are mentioned and shown throughout the text of this thesis.

Expressions, Values, and Variables The Scala syntax for simple expressions is similar to the syntax used in languages like C, Java, etc. For example `1 + 1` is a simple addition expression. Also, calling a function that has a return type and takes three arguments, such as `f(a, b, c)` is an expression.

One can assign a name to an expression `exp` via the syntax **`val`** `name = exp`. These names cannot be reassigned. One can also define a *variable* to save the result of `exp`, using the syntax **`var`** `v = exp`. Variables can be reassigned. Scala is a statically scoped language, i.e. the binding of a variable can always be determined by studying the program text and is independent of the run-time function call stack.

Scala also offers conditional expressions of the form **`if`** `cond_exp` **`then`** `exp1` **`else`** `exp2` with the expected semantics. For pattern matching, Scala uses the syntax

```
exp match {
  case p1 => exp1
  ....
  case pn => expn
}
```

These match expressions are evaluated from top to bottom, i.e. the right-hand side expression for the first pattern within a **`case`** that matches is executed. Only this **`case`** is executed, there is no "fall-through-semantics". One is not forced to specify match-expressions so that every possible case is covered. However, if during the execution of a match-expression no **`case`** matches for the given expression, Scala throws a matching error at runtime (Scala also features exceptions with a similar syntax as in Java). Case patterns may use wildcard patterns to avoid naming sub-patterns. For example, **`case`** `h :: _ => h` is a pattern for a list with at least one element (:: being syntactic sugar for a `Cons` constructor). It binds the name `h` to the head of the list, but uses the wildcard name `_` for the tail, i.e. the tail of the list is not bound by a name.

Like in Java, expressions may be grouped in a block via the syntax { `exp1`; `exp2`; ....; `expn` }. The result of such a block is always the result of the last expression. If every expression within a block is written on a new, separate line, Scala allows for dropping the semicolon between the different expressions. Scala also supports imperative language elements (such a variable assignments and standard while-loops, which have `Unit` as return time (the empty type).

Functions and Methods Being a *functional programming language*, Scala offers convenient syntax for declaring anonymous functions: For example, the expression

```
(a: T1, b: T2) => exp
```

defines an anonymous function that takes two arguments (of two types T1 and T2) and returns the result of executing exp. Function expressions can of course also be the arguments or return types of other functions and methods, as common for functional programming languages.

Methods are defined by specifying first a method signature and then an expression that defines the method. For example,

```
def methodname(a: T1, b: T2): R = exp
```

defines a method that takes two arguments of two types T1 and T2 and returns a value of type R. It is defined by expression exp. Methods are not expressions. They can only appear within classes, traits, or objects (see next paragraph).

Classes, traits, and objects Scala is also an *object-oriented programming language*, i.e. all code is always defined within an object or a class. During execution, only objects and instantiations of classes interact with each other. Standard classes as in Java are defined via the syntax

```
class Classname(a: T1, b: T2) {
  ... //(values, variables, methods)
}
```

Classes may be *instantiated* via the syntax new Classname(a, b), passing concrete arguments to the constructor parameters of the class. This syntax returns an object of type Classname, which may be assigned to a **val** or **var** with name objName. Then, one can access (public) field and methods of the class Classname via the dot-operator, as in Java (e.g. objName.methodname(...)). Note that fields and methods of class declarations in Scala are public by default. One can declare them as **private** or **protected** (with the same semantics as in Java) by adding the respective keyword in front of a field/method declaration.

Scala supports the (single) inheritance of classes, i.e. a class declaration may *extend* another class declaration (via keyword **extends**), inheriting all its fields and methods. A subclass may choose to *override* the fields or methods of its superclass by explicitly adding the **override** modifier to the new declaration in the sub-class.

Case classes are a special type of classes in Scala: Case classes are immutable and compared by value, unlike standard classes that are compared by reference. That is, two instances of a case class with the same constructor arguments are always considered equal, which is not the case for two instances of a standard class. Case classes are instantiated without the new keyword and automatically provide fields for their constructor arguments with the same names as the given class parameters. For example,

```
case class MyCase(a: T1, b: T2) {
  ... //(values, variables, methods)
}
```

defines a case class that one may instantiate by writing MyCase(e, f), and then one may access the constructor parameters of that instance via MyCase.a and MyCase.b.

In a standard class, every field and method declaration has to have a definition. If one wants to leave a field or method undefined, one has to either declare the class as **abstract** by adding this keyword in front of **class**, or declaring the class as a *trait* instead by writing **trait** instead of **class**. The main differences are that firstly abstract classes may have constructor parameters just like classes, while traits may not have any constructor parameters. Secondly, traits may be used as *mixins*, abstract classes can only be extended via the mechanism of standard (single) inheritance.

Traits are similar to Java's interfaces. In Java, a class may also only extend a single class, but implement multiple interfaces. Similarly, in Scala, a class may only extend a single class, but mixin multiple traits. As opposed to Java interfaces, traits may contain method definitions. In case that there are multiple definitions for a method with the same signature in the traits that are mixed into a class declaration, the Scala compiler *linearizes* the trait hierarchy, avoiding the diamond problem that classically arises in settings where multiple inheritance is allowed. For more information on how this linearization works, please refer to Odersky's bookk [OSV11].

Finally, one may define **object**s, which are classes that can only ever have a single instance. Fields and methods within an object are simply accessed via the object's name.

Implicits Scala supports implicit parameters, implicit field definitions, implicit methods, and implicit classes, often summarized as *implicits*. Classes, method parameters, field and method definitions may all be declared as implicit by adding the keyword **implicit** in front of the appropriate definition/declaration.

Implicit parameters are method parameters that users do not need to pass *explicitly* to a method. Rather, the compiler *implicitly* derives an appropriate argument for an implicit parameter. Note that this behavior is different from the behavior of *default* parameters, which Scala also supports: Method declarations may include default values for the last n parameters of a method. If a method call does not pass any value for these parameters, the call uses the declared default value. Implicit parameters allow for an even greater flexibility than default parameters: If a method parameter is defined as implicit, the compiler will look for accessible implicit field and method definitions which may be used to produce a value of the correct type.

Furthermore, the compiler may use any *implicit field definition* or *implicit method* accessible within the current scope for *implicit type conversions*: If a value at some position does not have the type required at this position, the compiler may call an implicit definition to convert the value to the required type. Similarly, the compiler may instantiate an *implicit class* if one attempts to access an object method that the current object does not provide: The compiler will look for an implicit class that can be constructed from the current object and has the appropriate method, create an instance from it and call this method.

All implicits may also be called explicitly by developers. Some Scala IDEs, such as IntelliJ by JetBrains, may also "unroll" automatically the implicits used within an expression, which is useful for debugging.

Implicits are especially useful for building embedded DSLs: Scala allows for

dropping parentheses of arguments of method calls in some cases (i.e. writing a call as `f a` instead of `f(a)`). Additionally, Scala allows almost any character in method names (except for some that are reserved for Scala-internal purposes, and of course method names may not clash with existing expressions such as numbers). Hence, methods may be named "+", "%" etc. Together with the features provided by implicits, this allows for building expressive embedded DSLs. We will see an example in Subsection 5.1.1.

2.5. Automated Theorem Proving

Throughout this thesis, we use automated theorem provers (ATPs) and SMT solvers (satisfiability modulo theories): Both an *automated theorem prover* and an *SMT solver* receive an input problem as a set of logical definitions and/or logical axioms, together with a goal to be proved within this definition. The ATP or SMT solver then attempts to automatically prove or disprove the given problem. Historically, ATPs and SMT solvers differed in the kinds of problems they were targeting: ATPs typically worked on pure first-order logic problems, while SMT solvers targeted problems in first-order logic *plus* different additional theories such as integer arithmetic, algebraic datatypes, etc. Nowadays, ATPs and SMT solvers are very similar to one another and available automated provers often internally use representative tools from both research areas.

Both ATPs and SMT solvers internally use a vast number of automated search techniques and calculi, in combination with heuristics that decide which of the numerous methods and rules to apply to a particular problem (since there are typically many options at any given point during the search, and exploring all of them would be infeasible in practice).

The "Handbook of Automated Reasoning" [RV01] is a vast and detailed collection of the numerous methods and calculi that ATPs internally use. The ATP world (and also parts of the SMT solver world) mostly employs *refutation-based proving* via resolution: One first adds the negation of the goal to be proven as axiom and then translates the entire problem into a clausal normal form (disjunctive normal form, where all formulas are expressed as disjunctions of conjunctions $\bigvee_i \bigwedge_j \phi$). Next, different inference rules are applied on these clauses, the most prominent one being the *resolution* inference rule:

$$\frac{\phi \vee c \qquad \psi \vee \neg c}{\phi \vee \psi}$$

The internal methods attempt to rewrite clauses via variable *unification* until the resolution rule can be applied. Eventually, if the empty clause is derived, the original goal is satisfiable (i.e. there is an instance of all variables so that the goal formula is true, hence we can say the goal is *proved*). If it is not possible to derive the empty clause, the goal is unsatisfiable (hence disproved).

In theory, resolution in (classical) first-order logic is *complete*, i.e. eventually, with enough resources, one may always determine if a goal is satisfiable or unsatisfiable.

In practice, however, the search space for applying all possible combinations of inference rules is too large to be fully explored. Hence, one typically calls an ATP with a certain timeout so that running the ATP for a certain proof problem may be *inconclusive* (i.e. it is unclear whether the goal is satisfiable or unsatisfiable).

Both ATPs and SMT solvers typically use a vast number of additional techniques, including for example from counterexample generation, model checking, and constraint solving.

2.6. Interactive Theorem Proving

One can use an *interactive theorem prover* for developing mechanized proofs. In their "pure" form, interactive theorem provers require the user to spell out every single step within a formal proof by applying the necessary rules manually. The interactive theorem prover checks that every step is in accordance with the existing definitions and that every rule used is indeed applicable at the point in the proof where it is applied. The focus in the area of interactive theorem proving is primarily in ensuring the overall correctness of a proof, and not so much in verification automation. Still, today many interactive theorem prover include themselves a vast number of automated proof methods that users may apply, and even call ATPs for automatically solving certain sub-problems within a proof.

Most existing interactive theorem provers are what we call in this thesis *general-purpose theorem provers*: They target numerous verification domains, just like a general-purpose programming language targets multiple

In this thesis, we use two existing interactive theorem provers for baseline proofs: Isabelle/HOL, which is an established, classical interactive theorem prover, and Dafny, which is newer and originally meant as a programming language with verification support, but can, to some extent, be used like an interactive theorem prover. We briefly introduce the main features and syntax of both systems that we use in this thesis. In Chapter 3, we present concrete examples.

2.6.1. Isabelle/HOL

Isabelle [Isa18] is an interactive theorem prover that supports multiple different logics. Isabelle/HOL is the probably widest used instance of Isabelle, targeting higher-order logic (HOL). The original Isabelle book [NPW02] contains detailed information about Isabelle's syntax and semantics. The official Isabelle webpage under `https://isabelle.in.tum.de/` contains documents on individual features available within Isabelle.

Syntax for specifications To specify a program in Isabelle, a user first has to develop so-called *theory files* or simply *theories*. Theories act as "containers" for definitions, theorems and proofs, just like modules or class definitions in programming languages. Theories may import other theories. Theories start with *theory name*, followed optionally by *import* declarations. Then, all definitions, theorems, and proofs are wrapped in an *begin* and *end*.

One may introduce closed algebraic datatypes via the keyword *datatype*, specifying the name of the datatype, followed by a list of data type constructors with arguments separated by |. Note that in Isabelle, any application of a function *f* or of a datatype constructor is always noted in the syntax *f a b* rather than the maybe more common *f(a,b)*. One may introduce an *underspecified* datatype in Isabelle via type parameters, always noted as *'a* for a type parameter named "a". For example, the type *'a list* is a list of elements of the underspecified type *'a*. It may be used with different concrete types. One may also abbreviate a longer type via the *type_synonym* keyword, with which one may also define a type synonym that is parametric in different types. Isabelle/HOL offers a large library of pre-defined data types, among which are *'a list* (for lists with elements of type *'a*), and *'a option* (for specifying that functions may fail, i.e. constructor *None* models failure, while *Some 'a* models the successful computation of an expression of type *'a*).

For formalizing definitions and functions, Isabelle offers a number of constructs: There is the simple construct *definition* (followed by a name, a type signature for the definition, keyword *where*, and then exactly one equation that specifies the definition). Definitions are similar to macros in some programming languages: They serve for defining a, possibly parametric, abbreviation for an expression. One cannot use *definition* to specify a recursive function (it is not allowed to introduce a recursive call within the definitional equation of a *definition*).

For specifying a recursive function, one may use keyword *primrec* (followed by a name, a type signature for the function, keyword *where*, and then one or more equations that specify how the function behaves for different argument patterns, separated by |). The *primrec* construct can only be used for specifying *primitive recursive* functions, i.e. functions whose recursive structure strictly follows the recursive structure of an argument defined via an algebraic datatype. For these functions, Isabelle is always able to automatically prove the termination of the function (Isabelle requires that all specified functions always provably terminate). For specifying functions with more complex recursive structures, one may either use *fun* (which attempts to prove termination automatically, using different automated methods to prove function termination - if it fails, users have to add hints via measure terms to help finding a termination proof) or *function* (which requires users to provide a proof of termination for the function entirely manually). The specification of functions and definitions always automatically produces internal rules and facts about the function/definition that can be used in proofs (e.g. induction rules).

On the right-hand side of a function equation, one may use language expressions common in programming languages: Firstly, *if c then e else f* can be used for conditional branching. Next, *let v = e in exp* can be used to bind the result of an expression *e* to a name *v* in expression *exp*. Finally, Isabelle also supports case expressions for matching on different expression patterns:

case exp of
$c_1 \Rightarrow e_1$ |
... |
$c_n \Rightarrow e_n$

Such an expression allows for matching an expression *exp* on its constructors (including patterns for the constructor arguments) c_1 to c_n, returning a different expression for each case.

One may specify inductive definitions (such as for example the typing rules of a type system) via keyword *inductive*, followed by a name, a type signature for the inductive definitions, optionally a pattern for a syntactic abbreviation/syntactic sugar for the induced inductive predicate, keyword *where*, and then a list of inference rules, separated by |. Each inference rule starts with a name for the rule, followed by a list of premises (several premises are contained in ⟦ ... ⟧, separated by semicolons), $\Longrightarrow$, and a conclusion. Just as is common for inference rules, all free variables within the rules are implicitly universally quantified.

Finally, there are different syntactic options within Isabelle/HOL for specifying theorems and lemmas. In this thesis, we use the following variant (from the Isar proof language, see next paragraph):
theorem name:
assumes $name_1$*:* $prem_1$
...
assumes $name_n$*:* $prem_n$
shows conc
This syntax defines a theorem with name *name* and n premises, each with their own name to which the proof below the theorem may refer. Finally, the conclusion *conc* of the theorem is preceded by keyword *shows*. Similarly, one can specify a lemma by using the *lemma* keyword instead of *theorem* above.

Syntax for developing proofs Each theorem and lemma specification within Isabelle has to be followed by a proof, consisting of proof commands and/or tactic applications. Isabelle offers two alternative styles for developing proofs: the so-called "apply-style" and the Isar proof language [Wen02]. The "apply-style" simply consists of a series of tactic applications, applying pre-defined rules and methods on a proof goal to modify it, split it into different goals, and to ultimately discharge (i.e. prove) it completely. Tactics may also consist in applying powerful proof search methods or external ATPs. Pure "apply-style proof scripts" are typically relatively short, but the resulting proof is not human-readable. Isar is a language consisting of various commands that remind a human reader of sentences that typically appear in manual proofs on paper ("Assume x, then show y", "For case a assume b, from this have c, then show d" etc.). Isar proof scripts are typically much more verbose than pure "apply-style scripts", but much more comprehensible for human readers, even if they have little knowledge of Isabelle. Both proof styles may also be mixed: For example, a proof script may be an Isar script for the top-level proofs, but for different parts within the Isar script where again a separate, inner proof block is required, one may use a shorter apply-style script. In this thesis, we use a mixture of both styles as just described, resulting in human-readable top-level proofs.

Isar proofs start with keyword *proof*, followed by the name of a top-level rule to apply to the top-level goal (e.g. an induction rule). Every Isar proof block always has a set of top-level assumptions that may be used via keyword *assume*, followed

by the assumption, and a conclusion which ultimately has to be proven via the *show* keyword (followed by the conclusion and a proof block). A conclusion may be abbreviated via the internal variable *?thesis*.

One introduces the proofs of different cases via the *case* keyword followed by the case name, followed by another Isar proof block that has the case assumptions and the specific conclusion for this case as assumptions and conclusion. Proofs of different cases are separated via keyword *next*. One may introduce any intermediate proof steps via keyword *have*, followed optionally by a name for the step, a formula, and a proof block that proves this formula. One may introduce a proof variable via keyword *obtain*, followed by a name for the variable, keyword *where*, and a (optionally named) formula constraining the new variable, plus a proof block proving this formula.

Proof blocks that consist entirely of rule applications (apply style) may be abbreviated with the *by* keyword followed by the list of rules and tactics to be applied. One may include previous assumptions and intermediate steps into a proof by preceding a *have*, *show*, *obtain* clause with keyword *from* followed by the list of the names of the facts to be used. Alternatively, one may use keyword *using*, followed by a list of names of facts, which has to appear after the formula to be proved, but right before the following proof block. Isar also offers different syntactical abbreviations for including the previous fact into a proof without explicitly naming it (e.g. the keywords *then* and *with*). Keyword *qed* concludes a proof block.

2.6.2. Dafny

Dafny is a programming language with verification support developed by Microsoft Research [Lei10]. Unlike Isabelle, Dafny originally is no fully-fledged interactive theorem prover, but rather a "verification-aware", imperative programming language. As such, it contains means for annotating functions with pre- and postconditions and loops with loop invariants, which Dafny tries to prove automatically. For more complex proofs, there are means to specify lemmas and to direct proofs via proof commands. The language for proof commands is intentionally syntactically identical to the programming language itself, in the spirit of the Curry-Howard isomorphism (programs are proofs and vice versa). For writing proofs in Dafny, one can either write scripts in the Dafny language with a very low granularity, relying very little on the internal verification automation. Or one can give only a few hints for the proof and let the gaps be filled by the verification automation. Of course, the more complex a proof is, the more "hints" one has to give in order for the automation to be successful.

Specifications To define types, one may either use construct **type** to define a type abbreviation, or **datatype** to define an algebraic datatype (where the data type constructors are separated by |). Underspecified type parameters may be introduced by enclosing their names in < ... >. One may use keyword **function**, followed by a name (optionally with a list of type parameters), a list of named function arguments with their types, and the function's return type, and finally a block that defines

the function. One may introduce a predicate via keyword **predicate**, followed by the predicate's name (optionally with a list of type parameters), and a list of named predicate arguments with their corresponding type, finally with a block that contains a logical formula.

From Dafny's syntax for function expressions and commands, we use the following constructs in this thesis:

- **match** with a list of cases (keyword **case**) for matching an expression on different patterns
- **if** ... **then** ... **else** for conditional branching
- v.constructorname? for checking whether a variable v (that has an ADT as type) is of the shape constructorname

We also use two data structures pre-defined within Dafny: seq<A> for sequences with elements of type A and Option<A> for optional values of type A. Sequences in Dafny support special syntax for accessing API functions. E.g. |s| is used to refer to the length of a sequence s and s[i] to refer to the i minus 1'th element of a sequence.

Formula expressions within Dafny use && for logical conjunction, || for logical disjunction, ==> for logical implication, and $\forall$ and $\exists$ for universal resp. existential quantification.

For predicates and functions, one uses keyword **requires** plus a formula for specifying a pre-condition, and **ensures** plus a formula for specifying a post-condition. Predicates and functions may have multiple pre- and post-conditions.

Proofs As mentioned above, proofs in Dafny use the same syntactical constructs as Dafny's specification language. Unlike specification constructs, proof blocks in Dafny may have "gaps" which indicate that at this point in the proof, the automated proof methods jump in in the background to fill this part of the proof, invisible to a human user.

One may introduce lemmas via keyword **lemma**, using the same syntax as for specifying predicates. In fact, lemmas are "ghost methods": Their declaration is ignored by the Dafny compiler and only relevant to the Dafny verifier. Premises and conclusions of a lemma are introduced via **requires** and **ensures**, just like pre- and post-conditions of functions and predicates. Next, lemmas require a proof block.

A **match** construct in a proof block stands for an induction or a case distinction, listing the different cases via **case**. Applying an induction hypothesis corresponds to a recursive call to the lemma in Dafny, passing the appropriate arguments to the lemma name to instantiate the induction hypothesis as needed. Similarly, one introduces a lemma application into a proof by calling the appropriate lemma like a method, passing arguments to instantiate the lemma. An **if** cond exp1 **else** exp2 expression (note that **then** is missing, so there is a small difference to the Dafny programming/specification language here!) stands for a boolean case distinction. One may introduce additional variables into a proof via keyword **var** (which also exists as imperative construct within the Dafny programming language).

Chapter

3

Survey: Type Soundness Proofs for DSLs with Existing Provers

We start by analyzing the target verification domain of this thesis: mechanized type soundness proofs of DSLs. To this end, we first look more closely at the mechanization and automation of soundness proofs of general-purpose type systems, summarizing the mechanization efforts within the PoplMark challenge that we already briefly introduced in Subsection 1.1.2. We describe the main obstacle for the automation of these proofs: the "name-binding problem". We use our observations to restrict the set of DSLs we are interested in accordingly (notably, focusing on DSLs without first-class binders), arriving at a subset for which it seems feasible to attempt full automation (Section 3.1).

Next, we choose two established verification systems, Isabelle/HOL [Isa18] and Dafny [Lei10], and investigate how one can presently mechanize a soundness proof for the type system of a DSL that meets our criteria within these two systems. We describe the specifications and proofs in both systems, focusing on how much of these proofs can be discovered automatically within these two tools and how much skillful user interaction is required (Section 3.2 (Isabelle) and Section 3.3 (Dafny)). Afterwards, we discuss the pros and cons of using further possible verification systems with regard to the degree of automation they can achieve for the kind of type soundness proofs we are interested in (Section 3.4). We also discuss how one could use or extend existing verification systems for the automation of our target verification domain. In our discussion, we focus on how far one could automate such proofs using existing systems, and on how easy it would be for other developers to use and extend the resulting augmented verification systems.

Remark 3.1. The author of this thesis co-published an early version of the SQL Dafny proof that is introduced in Section 3.3 in a workshop paper [GEM16]. Apart from this publication, most of the content of this chapter was unpublished before the submission of this thesis. ◊

3.1. Target Languages: Motivation and Clarification

When studying the solutions to the PoplMark challenge in detail, one can quickly see that the main obstacle researchers faced during the mechanization of the challenges was and is an issue that is often called the "name-binding problem". Finding a good solution to this problem ultimately became the main focus within the challenge, as also described in the latest version of the challenge itself [Ayd+05]. We describe the "name-binding problem" in detail in Subsection 3.1.1 and argue why this problem is a main hurdle for full automation of type soundness proofs of general-purpose languages.

Based on our insights from studying the research efforts within the PoplMark challenge, we narrow the focus of typed programming languages we consider to DSLs with certain characteristics (basically avoiding the "name-binding problem" (Subsection 3.1.2)). We argue why we believe that automating type soundness proofs for languages from this set is feasible. Finally, we informally introduce an example DSL (a subset of typed SQL) which we are going to use as a concrete example in the rest of this chapter and later in this thesis (in Chapter 7) as a case study for our own system (Subsection 3.1.3). We motivate the choice of the example DSL and of the features it supports.

3.1.1. The "Name-Binding Problem"

We study in more detail why mechanizations of soundness proofs of type systems for general-purpose languages are so hard. We focus on the work achieved so far within the PoplMark challenge (see Section 1.1.2).

Looking at canonical texts that treat the untyped and typed λ-calculus and proofs of progress and preservation on paper, we often find conventions such as:

> "Terms that differ only in the names of bound variables are interchangeable in all contexts." (Convention 5.3.4 in Pierce TAPL [Pie02])

This convention intuitively explains the notion of *alpha equivalence*: If we have a syntactical construct in a language that introduces a variable binding (or "name binding"), the concrete name of the variable is irrelevant. In the simply-typed λ-calculus, we have the abstraction construct $\lambda x : T.t$, where t is a term in the simply-typed λ-calculus that may or may not contain variable x, representing a function that takes an argument x of a type T. When applying $\lambda x : T.t$ to a term a, all occurrences of x in t are substituted with a (or a reduced version of a, depending on the evaluation strategy (call-by-value, call-by-name etc.)). The original lambda-term is then reduced to the term that results of this substitution.

Hence, it does not matter how the binder of the function is called: A term $\lambda x : T.t$ is alpha-equivalent to a term $\lambda y : T.t'$, where t' is a variant of t where all bound occurrences of x have been replaced with y. The same resulting expression is computed for a given argument no matter which of these λ terms ist used.

When defining the behavior of function applications, we need to formally define the notion of "substituting" a term within a λ-abstraction. Here, we have to make

sure to neither accidentally erase variable bindings (e.g. $[x \to y](\lambda x : T.x)$ should *not* become $\lambda x : T.y$), nor to accidentally let a free variable become a bound one, called *variable capture*. The latter means that $[x \to z](\lambda z : T.x)$ should *not* become $\lambda z : T.z$.

The solution to the first problem is simple: The substitution function simply has to stop substituting when encountering a λ abstraction which binds again the same variable the function is currently substituting. Variable capture, however, is slightly more difficult to resolve. On paper, one can simply come up with conventions such as the one presented above and be done with it, allowing for implicit renaming of bound variables whenever needed during substitution. The same can be done within manual progress and preservation proofs, when one faces arguing about properties of terms where bound variables have to be renamed during substitution: Any property that was already known to hold for the term before the renaming automatically holds for the renamed version due to alpha equivalence.

However, when developing a mechanized version of a concrete formal specification or implementation of the substitution function, one has to be explicit regarding the treatment of names: All bound variables whose names clash with free variables within the term on which a substitution is applied have to be renamed with "fresh" variable names. "Fresh" names are names that neither occur in the term itself, nor in the argument to be substituted. For example, to calculate $[x \to y\ z](\lambda y : T.x\ y)$, one has to first rename the bound y in the λ-term, e.g. to $\lambda w : T.x\ w$, and then calculate $[x \to y\ z](\lambda w : T.x\ w)$, giving $(\lambda w : T.y\ z\ w)$.[1] A substitution function that actively avoids variable capture is called a *capture-avoiding substitution*.

Of course, if such a renaming is explicitly implemented, one has to face numerous situations within a mechanization attempt of a soundness proof where the renaming "gets in the way": A property is given for a term with its original names for bound variables, but has to hold for a term where some bound variables might have been renamed. Since such terms are syntactically different, most provers also treat them as different, so one would be stuck in the proof at this point. For example, when proving preservation for the application case within the type soundness proof of the simply-typed λ-calculus, one has to use an auxiliary lemma called "substitution lemma" in the literature:

Lemma 3.1. *If* $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash s : S$, *then* $\Gamma \vdash [x \to s]t : T$. $\Diamond$

One proves this lemma classically via an induction on a derivation of the statement $\Gamma, x : S \vdash t : T$ (see for example Pierce TAPL [Pie02], pp. 106 and 107). In the case where $t = \lambda y : T_2.t_1$ and $T = T_2 \to T_1$, the conclusion of the induction hypothesis gives us $\Gamma, y : T_2 \vdash [x \to s]t_1 : T_1$, which we need to use to argue that the conclusion of the abstraction case holds. However, in case y was renamed during substitution, e.g. to w because s contained y as free variable, we would instead need $\Gamma, w : T_2 \vdash [x \to s]t_1' : T_1$, where t_1' equals t_1 with the bound occurrences of y replaced by w. Without doing anything else, we simply would not obtain this statement within our mechanized proof attempt and be stuck.

[1] example taken from Pierce TAPL [Pie02], p. 71

In order to advance, either a notion of alpha equivalence would have to be built in into the verification system explicitly, or some other solution would have to be found for dealing with bound variables and avoiding variable capture during substitution.

This issue, often called the "name-binding" problem, was quickly discovered to be one of the main issues within the PoplMark challenge. The latest version of the PoplMark challenge document [Ayd+05] states:

> "The problem of representing and reasoning about inductively-defined structures with binders is central to the PoplMark challenges."

There are several different strategies on how to deal with the "name-binding" problem within the PoplMark challenges. The different solutions to the challenges differ in how they approach this problem. We summarize the most important solutions here, referring to the already mentioned webpage of the PoplMark challenge for the remaining approaches (see Section 1.1.2).

De Bruijn's nameless representation One popular and in implementation practice very efficient solution uses *De Bruijn*'s nameless representation for variable binders [Bru72]. In this representation, variable names do not exist, and variables are referred to via indices that point directly to their binders. That is, a 0 points to the innermost λ, a 1 to the next λ seen from the inside, and so on. Free variables get assigned indices that do not clash with the indices of bound variables. For example, the named term $\lambda x.\ \lambda y.\ x\ (y\ x)$ in the untyped λ-calculus corresponds to $\lambda.\ \lambda.\ 1\ (0\ 1)$. (In the simply-typed λ-calculus, the binders would retain the argument type in addition.) This nameless representation yields a unique representation of every λ term, hence there do not occur any issues with reasoning about alpha-equivalence in proofs. Also, this representation is convenient for storing terms in memory.

On the other hand, De Bruijn's nameless representation requires lots of auxiliary functions that correctly shift variable indices during substitution, which can be tricky to get right. Progress and preservation proofs for languages using the nameless representation also require a lot of auxiliary lemmas regarding shifting and substitution, which blow up the proofs considerably.

Both Vouillon's PoplMark solution in Coq [Vou12] and Berghofer's solution in Isabelle/HOL [Ber07] apply De Bruijn indices, the management of which makes up a large part of the mechanized proof. Vouillon managed to separate the low-level lemmas about index shifting etc. as much as possible from the main definitions and proof development. But still, these lemmas are there and have to be fully proven in order to obtain a completely machine-checked formalization of the PoplMark challenge with this representation.

Locally nameless representation A variant of De Bruijn's nameless representation is the "locally" nameless representation: Here, variables bound by λ-expressions are represented via De Bruijn indices, but free variables have symbolic names. This style improves the readability of terms compared to the "purely" nameless representation by De Bruijn, while still retaining the main advantage of De Bruijn's

representation: Alpha equivalence is term equality. On the other hand, the locally nameless representation does not decrease the overall "boilerplate development" in specifications and proofs needed for De Bruijn's representation. Rather, it increases the amount of auxiliary definitions and lemmas: In addition to the index shifting required for the purely nameless representation, one requires also a substitution function for named variables, along with the corresponding auxiliary lemmas. Hence, regarding proof effort, one does not "win" anything when using the locally nameless representation instead of pure De Bruijn.

Within the PoplMark challenge, the locally nameless representation was used by several people, notably within Coq: Firstly, Leroy [Ler07] uses the locally nameless representation. He comments that "[...] the development is larger than a pure De Bruijn solution"[2]. Influenced by Vouillon [Vou12] as well as by Leroy [Ler07], Charguéraud [Cha12] also developed a partial PoplMark solution within Coq using the locally nameless representation. Charguéraud notes that the overall development becomes more "intuitive" when using the locally nameless representation compared to pure De Bruijn, but also acknowledges that the proof is "slightly more involved" in some cases than the purely nameless representation.

Higher-order abstract syntax An entirely different approach to deal with the "name-binding problem" than the nameless or locally nameless representation is using *higher-order abstract syntax* (HOAS). This approach is based on having a core meta-language with binders which is known to be sound with regard to different behavioral problems. Next, one encodes the object language in question (e.g. for the PoplMark challenge, System $F_{<:}$) using variables in the meta-language to encode variables in the object language, and binding constructs from the meta-language to encode binding constructs in the object language. Thus, capture-avoiding substitution etc. does not have to be defined for the object language. Instead, the trusted implementation of the meta-language is used implicitly in the background. Also within progress and preservation proofs, it is not necessary to argue about alpha-equivalence issues, since the proofs implicitly rely on corresponding proofs from the meta-language.

Thus, HOAS basically circumvents the entire "name-binding issue" by deferring it to the sound meta-language, which is of course a huge advantage for any particular proof problem. On the other hand, however, it can be relatively hard to come up with adequate encodings of certain advanced object-language features within the meta-language, or also of proof steps (e.g. how one can "isolate" a variable in the middle of a context, which is needed within the PoplMark challenge to proof transitivity of subtyping).

Complex interactive theorem provers like Isabelle [Isa18] and Coq [Tea19] do not provide such a core meta-language which one could apply for the HOAS approach. Consequently, there is no HOAS solution to the PoplMark challenge done in Coq or Isabelle. An established theorem prover which provides the well-understood dependently-typed λ-calculus as meta-language is Twelf [PS99]. The aforementioned

[2] `https://www.seas.upenn.edu/~plclub/poplmark/leroy.html`

Twelf solution of PoplMark by Ashley-Rollman, Crary, and Harper [3] naturally uses the HOAS approach. As said, the authors report that while using HOAS simplifies the "name-binding problem", it complicates other aspects of the proof. Furthermore, using the HOAS approach requires understanding an "unorthodox" way [4] of theorem proving (concretely, proving theorems using a logical framework [Pfe91], which essentially exploits the Curry-Howard isomorphism between proofs and programs).

Nominal logic The last approach we discuss here attempts to create and use a proving environment that is as close as possible to pen-and-paper-reasoning with regard to the "name-binding problem": *Nominal logic* [Pit03] encapsulates all issues about alpha equivalence within the logic underlying the prover. Nominal logic extends the usual axioms of classical first-order logic with "swapping" operations which can interchange names in a term, a freshness relation that expresses that a given atom is fresh with respect to a term, and a number of axioms and "specialized" quantifiers that express properties on these structures. Using these structures, one can define special nominal induction principles. Within nominal logic, one can define datatypes composed of "alpha-equivalence classes", i.e., alpha-equivalent terms are treated as equivalent within the logic.

In an abstract way, the idea behind nominal logic is similar to the idea behind the HOAS approach: Any "name-binding issues" are deferred to the underlying "structure" (meta-language or logic) used for specifications and for proofs. Hence, using nominal logic shares the advantages of the HOAS approach, seen from a general perspective. The main advantage nominal logic has over the HOAS approach is that one can encode specifications and proofs in "conventional" logic and does not have to understand logical frameworks and to come up with "unorthodox" encodings of language features. However, looking a little more closely, obtaining the built-in notion of alpha-equivalence in nominal logic does not come entirely for free, but requires users to prove that the axioms on which the logic is based hold when defining and using their own "nominal" datatypes.

Nominal logic is being implemented within Isabelle/HOL by Urban et al. [UT05] as "Nominal Isabelle" since 2005, inspired by work on the PoplMark challenge. It is now a large research project involving several different authors. The first version of Nominal Isabelle was only able to support binding constructs with one binder. Therefore, it was not possible to use Nominal Isabelle 1 for the more advanced parts of the PoplMark challenge involving records. Hence, the submitted solution to the PoplMark challenge only addressed the simpler parts of the challenge [5]. The second version of Nominal Isabelle strives to remedy this issue. As of today, it is still in a beta stage (the latest version of Nominal Isabelle is for Isabelle 2016), and there has not yet been an updated submission to the PoplMark challenge using Nominal Isabelle 2 that we know of. However, the latest concepts within Nominal Isabelle 2 are promising, and it remains to be investigated how much user effort

[3] `https://www.seas.upenn.edu/~plclub/poplmark/cmu.html`
[4] from the point of view of people working outside the area of programming languages
[5] `https://www.seas.upenn.edu/~plclub/poplmark/urban.html`

is involved in using the updated logic from Nominal Isabelle 2 for type soundness proofs.

Summary Dealing with the "name-binding problem", which we explained in detail at the beginning of this subsection, is necessary for mechanizing soundness proofs of general-purpose programming languages. A solution to the problem, ideally an automatic one, is definitely needed first before attempting to automate soundness proofs of type systems for general-purpose languages. However, addressing the "name-binding problem" within theorem provers is not trivial. To date, several good solutions exist, the most important of which we briefly presented here. Most of these solutions require extensive boilerplate specifications and lemmas (nameless representation/locally nameless representation, nominal logic), or a certain amount of user creativity (HOAS). The development of these solutions took a lot of time and energy and is not yet fully complete, which probably prevented further research efforts on automating the mechanization of the PoplMark challenge. Automating one or more of the existing approaches on "name-binding" seems possible, but is a challenging research topic on its own.

3.1.2. Focusing on "Simple" DSLs

In this thesis, we focus on soundness proofs of type systems for DSLs. As mentioned already in the Introduction, the motivation for this focus is twofold: On the one hand, the development and usage of DSLs is now more and more wide-spread, but resources for adding type systems to these languages and for ensuring their soundness are often scarce. On the other hand, the overall complexity of type soundness proofs for DSLs is likely to be lower than for general-purpose languages, raising the chances of obtaining a high degree of automation for such proofs.

However, arbitrary DSLs may of course be as complex as any general-purpose language, depending on which language features the DSL supports. Having studied and elaborated in the previous subsections which difficulties arise when mechanizing or automating soundness proof of general type systems, we narrow the focus for the DSLs we consider in this thesis. Our goal is to arrive at an interesting subset of type systems for DSLs for which automation of soundness proofs seems feasible today with a reasonable effort.

We consider in this thesis what we call simple DSLs.

Definition 3.1 (Simple DSLs)**.** *Simple DSLs* are DSLs which do not contain any first-class binders (neither on the language and nor on the type level). ◊

Put differently, the definition above forbids DSLs with any language constructs the semantics of which requires defining substitution functions that rename variables during reduction and/or during typing. This restriction does not include language constructs that contain static names, provided the language's semantics and type system ensure the uniqueness of such names within terms. For example, we may define language constructs for introducing stored variable references, provided that we ensure the uniqueness of variable names via the reduction semantics as well

as within the type system to avoid shadowing of variable names (which may be problematic in soundness proofs). Similarly, we may introduce other domain-specific constructs with static names that never need to be substituted, e.g. table names within database query languages.

By narrowing the focus of the DSLs we consider strictly according to Definition 3.1, we avoid the issues arising from the "name-binding problem" described in Subsection 3.1.1 entirely. The restriction on simple DSLs explicitly forbids a large number of different constructs that explicitly or implicitly introduce named abstractions into expressions. This includes of course anything similar to a λ-abstraction, but also for example object creations and method invocations such as present in core calculi of Java (e.g. Featherweight Java [IPW01a]): Creating an object introduces object methods with parameters, the usage of which requires method invocation, whose definition in turn requires substituting arguments into method bodies.

The restriction on "abstraction-free" languages implies that simple DSLs are also free of a number of other advanced language and type-system features which complicate type soundness proofs and mostly only make sense in combination with abstraction constructs. To name the two main features that are implictly excluded (the exclusion of which in turn excludes a number of even more advanced features):

- Full subtyping: The ability of a type system to treat a specific type as a more general type depending on the context in which it appears is called *subtyping* (see e.g. Pierce's TAPL [Pie02], Part III). This flexibility during typing allows a type system to be less restrictive when typing abstractions that can interact with different arguments of "similar" types. The canonical example for this is λ-abstractions that take records as arguments: Such abstractions can operate with any record argument that has *at least* as many fields as the abstraction's body accesses, but may have more. Full subtyping only makes sense if there is a language construct that may abstract a certain part of a program's behavior and interact with many different arguments (of different types) during expression reduction. If we do not have such abstraction constructs, a type system may always assign a single type to every construct within an expression without being overly restrictive.

- Polymorphism: Abstractions may not only abstract over expressions, but also over the types of expressions. This is a further level of abstraction, called *polymorphism*, which, like subtyping, allows a type system to be less restrictive during typing. As an example, consider a typed λ-abstraction that represents the identity function, $\lambda x : T.x$. Without polymorphism, applications of this abstraction can only ever be typed with a single argument type T. A type system that shall type-check applications with arguments of a different type needs to abstract over type T, i.e. to introduce a first-class binder on the type level, which would also require to specify substitution on the type level.

While the restriction to type systems of simple DSLs definitely excludes a large number of interesting languages from our focus, it does retain several interesting

type systems for DSLs that are also relevant in the real world. For example, even though we exclude classical full subtyping, one may still include some "lighter" versions of subtyping in type systems of simple DSLs, such as for example allowing integer numbers in addition expressions to also type as floating-point numbers etc. However, in the absence of binding constructs, this "lighter" subtyping constitutes only an optimization of a type system: Instead of introducing subtyping, one might as well spell out all the different typing rules necessary for different combinations of types. This could even be done automatically. Hence, in this thesis we are simply going to assume that we always deal with a *syntax-directed type system* without any notion of subtyping, i.e. with a type system that has exactly one typing rule for each language construct.

By restricting ourselves to syntax-directed type systems of simple DSLs, we arrive at a set of type systems for which it seems feasible to automate type soundness proofs: Soundness proofs of such "simple" type systems are very structural and rather repetitive. Both for progress and for preservation theorems, one can always proceed via structural induction on expressions, obtaining one induction case per language construct. Proving the induction cases is "only" a matter of applying appropriate case distinctions and auxiliary lemmas (of which there may be quite a large number!) until one can apply the induction hypothesis (if needed).

Since we ruled out capture-avoiding substitution and hence the possibility of renamings that occur during expression evaluation, there is no risk of obtaining language constructs during the proof so that the induction hypotheses "do not fully fit". Furthermore, the structure of the auxiliary lemmas needed in progress and preservation proofs of "simple" type systems is very schematic as well: Basically, one needs just needs to "transfer" the progress and preservation theorems to any relevant combination of auxiliary function called within the type system and within the reduction semantics. We will expand these first ideas within the remainder of this thesis, notably in Chapter 6.

In the remainder of this thesis, whenever we talk about DSLs, we mean simple DSLs in the sense of Definition 3.1.

3.1.3. Example DSL: A Subset of Typed SQL

To concretize our survey of existing verification tools for mechanizing type soundness proofs for DSLs, we choose a concrete DSL that meets our requirements from Subsection 3.1.2. Beyond these requirements, the chosen DSL should be

- known and used in realistic applications
- statically typed or allow for being augmented with static typing
- have a reduction semantics that is specifiable as a small-step reduction semantics, i.e. the reduction of an expression proceeds by subsequent reduction of subexpressions, generating intermediate, partially reduced expressions along the way (as also used for languages described in Pierce's TAPL [Pie02]).

- be small enough to be "manageable" for mechanizing a type soundness proof in different existing tools
- be interesting enough to showcase the nature of different proof steps that are typical for progress and preservation proofs so that we can assess how easy the mechanization of such typical steps is within existing verification sytems
- have a type system whose soundness proof is hard enough to show that (partial) automation of the proof is desirable

Our initial example of a simple language with typed arithmetic expressions (see Section 2.2) meets some of our criteria, notably Definition 3.1 (no abstraction constructs), is statically typed, has a small-step semantics, is "small enough", and could even be argued to be known and used within realistic applications. However, its type soundness proof is quite small and only moderately "interesting": The proof does not require interesting auxiliary lemmas (beyond type inversion, also called canonical forms lemma, which is implied by a type system's definition). Hence, we resort to using typed arithmetic expressions as a running example in this thesis to explain basic concepts of our approach. However, for the purposes of the survey in this chapter as well as for our case studies (Chapter 7 and Chapter 8), we choose a slightly larger example DSL with more interesting proof cases.

As a concrete example for a DSL that fits our requirement from Definition 3.1 as well as the requirements just mentioned, we choose a subset of a typed variant of SQL [CB74]. SQL is a realistic language that is heavily used within database management. It can be used to query data that matches specific conditions from databases and to create and manipulate tables within databases, both from scratch and by merging existing tables.

SQL queries are traditionally not statically typed. Hence, SQL queries that access non-existent attributes or compare attributes of incompatible types fail at run time. In order to statically detect SQL queries that fail at run-time, one can add a type system for SQL queries. Full SQL is a relatively complex language with many constructs. But we can easily choose a subset of SQL that on the one hand is small enough to be "manageable", and on the other hand hard and interesting enough for proof automation.

We focus the subset of SQL we consider on

- projection (choosing columns) on single tables from a static table store
- selection (choosing rows) according to basic row predicates on single tables from a static table store (comparisons of numerical table values as well as basic boolean operators)
- set operations (union, intersection, difference) on the results of two queries that return tables with equal table schemas

That is, we leave out data manipulation (e.g. creation, deletion, or modifying of existing table entries), complex selection predicates, aggregation and grouping

features of SQL, nesting of SELECT FROM WHERE queries, as well as joins and cross-products of two or more tables. Most of these features could be added to our subset of SQL in a straightforward way, but would blow up the soundness proof of the corresponding type system considerably. For the purposes of this thesis, we focus on a small, but interesting subset of features, the concrete choice of which we motivate next.

We choose column projection and row selection (via the known "SELECT ... FROM ... WHERE ... queries", which first select rows from a table and then project columns on the result of the selection) since on the one hand, these features are basic features of SQL that are widely known and understood. On the other hand, implementing column projection and row selection from scratch using only basic data structures (e.g. lists of lists for modeling tables) is relatively complicated: One needs to define several basic auxiliary functions for locating a certain column by going through a table and for retrieving single columns from a matrix, for pasting them together to form a new table, etc. These functions are then part of the reduction semantics of our subset of SQL, hence one needs to extensively reason about their behavior within progress and preservation proofs. Thus, the SELECT FROM WHERE case of our subset of SQL is an example of a complex case within progress and preservation proofs where one needs to create and appropriately use a larger number of auxiliary lemmas. This is expected to be the case for larger and even more interesting DSLs, hence our example language should feature such a "larger" case.

On the other hand, since we exclude nesting of "SELECT FROM WHERE queries" for simplicity, the case for column projection and row selection does not have an induction hypotheses. However, in general soundness proofs of type systems for DSLs are expected to require induction and to have cases whose proof requires applying induction hypotheses. To have an example of such a case as well within our example DSL, we add set operations on query results: A set operation takes two queries, which can be SELECT FROM WHERE queries or also again set operations, and applies the outer set operation to the results of these two queries. Since there is an inner reduction of the sub-queries, one needs to apply the corresponding induction hypotheses when reasoning about the cases for the set operations within the progress and preservation proof of our subset of SQL. We expect the proofs of the cases for different set operations to be very similar. We choose several set operations in order to investigate whether this expectation holds.

3.2. Using Isabelle/HOL for Type Soundness Proofs

We specify the subset of typed SQL that we outlined in Subsection 3.1.3 in Isabelle/HOL [Isa18], using the Isabelle2018 distribution. We use Isabelle to prove progress and preservation for the type system of our subset of SQL. In this section, we describe both the specification of SQL and the progress and preservation proofs in detail, showing relevant excerpts of our Isabelle theory files. Our full Isabelle theory files are available at `https://bitbucket.org/cygne_noir/sql-isabelle/`

`src/master/`.

Later in this thesis, when we model our typed subset of SQL in other systems and input languages (see Section 3.3 and later Chapter 7), we shorten the description of the specification or proofs of our typed subset of SQL, referring to the present section for details. At the end of this section, we discuss the proof process itself (Subsection 3.2.3). Readers for whom the specification details of our typed subset of SQL in Isabelle are not relevant at the moment may safely skip ahead to Subsection 3.2.3.

3.2.1. Specifying SQL Semantics and Type System

We start by specifying the basic concepts needed for modeling SQL queries: tables and table environments. On top of these basic data structures, we model the syntax of our subset of SQL and its reduction semantics (as a small-step semantics). Next, we model types of tables and a type system for our subset of SQL.

Modeling Tables

Listing 3.1 shows the basic data structures that we use to model tables in Isabelle. We use Isabelle's lists (from the *Main* library) to model single rows of tables (line 2 in Listing 3.1). Then, we model "raw" SQL tables as lists of rows (line 3 in Listing 3.1). For better readability, we define type synonyms within Isabelle for the appropriate types of rows and "raw" tables. We use a type parameter *'val* to abstract over the concrete type of table values.

Tables typically have a header made from attribute names used to address individual table columns. We model table headers as lists, again abstracting over the concrete type of attribute names using type parameter *'id* (line 1 of Listing 3.1). With these basic ingredients, we define the datatype *Table*, which composes an attribute list (i.e. a table header) and a "raw" table (line 5 in Listing 3.1).

Later, for the specification of SQL's semantics, we will require underspecified functions operating on our underspecified table values. We define these by applying Isabelle's *locales* [Bal04]. We introduce two underspecified functions *gt* and *lt* as locale parameters, for comparing whether a table value is greater respectively smaller than another table value (lines 9 and 10 in Listing 3.1). During the rest of the specification of our subset of SQL, we specify functions on tables within the context of locale *Table* so that we have access to the underspecified functions *gt* and *lt*. One can argue about concrete instances of *gt* and *lt* by interpreting locale *Table*.

Next, we define a parametric data structure *Environments* in Listing 3.2 (line 1), which we use both for modeling stores of named tables as well as for storing named table schemas (later when we model the type system of our subset of SQL). We define a simple primitive recursive function *lookupEnv* to look up items within environments via their *'id* (line 3 to 8 of Listing 3.2). Note that we could also use Isabelle lists of pairs to model environments, but refrained from doing so in order to obtain a structurally simpler specification of *lookupEnv*. Note that, also for simplicity, we ignore any duplicates within environments during lookup. Function *lookupEnv* simply always returns the first table with name *n* within the given environment.

```
1  type-synonym 'id AttrL = 'id list
2  type-synonym 'val Row = 'val list
3  type-synonym 'val RawTable = ('val Row) list
4
5  datatype ('id, 'val) Table = table 'id AttrL 'val RawTable
6
7
8  locale Table =
9  fixes gt :: 'val ⇒ 'val ⇒ bool
10  and lt :: 'val ⇒ 'val ⇒ bool
11 begin
```

Listing 3.1.: Basic data structures for tables in Isabelle

```
1  datatype ('id, 'a) Env = empty | bind 'id 'a ('id, 'a) Env
2
3  primrec lookupEnv :: 'id ⇒ ('id, 'a) Env ⇒ 'a option
4  where
5  lookupEnv n empty = None |
6  lookupEnv n (bind m a e) = (if (n = m)
7    then (Some a)
8    else (lookupEnv n e))
```

Listing 3.2.: Environments for table/table type stores in Isabelle

This does not introduce any ambiguous behavior in this case, since our table store is "read-only" anyway: We do not model any constructs for extending the table store.

SQL Syntax

We define basic type synonyms and datatypes for the SQL constructs we sketched in Subsection 3.1.3, shown in Listing 3.3. The top-level datatype of queries is *Query* (line 14 to 18), which is parametric in a type for identifiers of tables and columns (attributes) *'id* and in a type for table values *'val*. The idea for our formalization of certain SQL queries is that query values will be tables, i.e. every query will ultimately be reduced to a table by the semantics we are going to define next. Datatype constructor *tvalue* (line 14) wraps table values. The three recursive datatype constructors *union*, *intersection*, and *difference* (lines 16-18) model queries for the three set operations we want to support; each set constructor takes two queries as arguments. In line 15 of Listing 3.1.3 we define the constructor for modeling the standard SELECT FROM WHERE queries from SQL, out of which we will model row selection using certain predicates as well as column projection on single tables. Constructor *selectFromWhere* takes

1. A list of attribute names, designating columns for projection. We will use the intermediate datatype *Select* (line 1 in Listing 3.3) to quickly distinguish

```
1  datatype 'id Select = all | list 'id AttrL
2
3  type-synonym 'id TRef = 'id list
4
5  datatype ('id, 'val) Exp = const 'val | lookup 'id
6
7  datatype ('id, 'val) Pred = ptrue |
8    And ('id, 'val) Pred ('id, 'val) Pred |
9    Not ('id, 'val) Pred |
10   Eq ('id, 'val) Exp ('id, 'val) Exp |
11   Gt ('id, 'val) Exp ('id, 'val) Exp |
12   Lt ('id, 'val) Exp ('id, 'val) Exp
13
14 datatype ('id, 'val) Query = tvalue ('id, 'val) Table |
15   selectFromWhere 'id Select 'id TRef ('id, 'val) Pred |
16   union ('id, 'val) Query ('id, 'val) Query |
17   intersection ('id, 'val) Query ('id, 'val) Query |
18   difference ('id, 'val) Query ('id, 'val) Query
```

Listing 3.3.: Data structures for modeling SQL queries in Isabelle

between projection on *all* columns (* in SQL) of a given table and between projection on a *list* of given attribute names.

2. A list of table names *TRef* (line 3 in Listing 3.3) for designating the table(s) on which the SELECT FROM WHERE query shall operate. Note that we are only going to support lists of length 1 (i.e. single table names) here, since we decided to omit cross-products and joins from the subset of SQL we are considering here (see Subsection 3.1.3). Nevertheless, we use a list of table names at this point in order to already enable future extensions of the considered subset of SQL.

3. A predicate for row selection, modeled via the intermediate datatype *Pred* (lines 7 to 12 in Listing 3.3). We support comparisons of table values against each other or against given constants: Equality via constructor *Eq*, "greater than" via constructor *Gt*, and "less than" via constructor *Lt*. These three comparison constructors each take two arguments of type *Exp* (line 5 in Listing 3.3), which in turn has one constructor for designating table value constants (*const*) and one constructor for designating values within a column referred to via an attribute name *'id* (*lookup*). Apart from value comparison, we support boolean conjunction (constructor *And*) and negation (constructor *Not*) in row selection predicates. Constructor *ptrue* allows for simply selecting every row (e.g. for using a SELECT FROM WHERE query for pure column projection).

SQL Semantics

We model the semantics of our subset of SQL as a small-step reduction semantics, using a primitive recursive function in Isabelle. Listing 3.4 shows an excerpt of this function. Function *reduce* models *single* reduction steps of queries. To fully reduce a query to a table, *reduce* might have to be called several times, depending on the query. Function *reduce* gets as arguments an SQL query and an environment that stores tables under table names (of type *'id*), i.e. a table store. The result of a call to function *reduce* is either another query (wrapped in *Some*) or *None* to indicate that the reduction of a query is not possible.

Table values (*tvalue*) cannot be reduced further (line 4 of Listing 3.4).

SELECT FROM WHERE queries (*selectFromWhere*) are processed as follows:

1. We check whether the list of given table names only consists of a single element, since we decided to not support anything else (line 6 of Listing 3.4). If the check is not successful, we fail (returning *None*).

2. If that is the case, we attempt to look up this single table in the given table store (line 8 in Listing 3.4). If the lookup is not successful, we fail (returning *None*).

3. If the lookup is successful, we first apply row selection with the given predicate *p* by calling the auxiliary function *filterTable* (line 12 in Listing 3.4).

4. We apply projection on the result of the row selection by calling the auxiliary function *projectTable* (line 13 in Listing 3.4). If projection is successful, we produce a table value with the resulting table (otherwise we return *None* to indicate that the reduction cannot proceed).

To explain how set queries are reduced, we focus on the *union* case in Listing 3.4 (lines 18 to 31): First, *reduce* checks in lines 19 and 20 whether both queries are table values already (the auxiliary predicate *isValue* only returns true for *tvalue* queries). If that is the case, *reduce* blindly produces a table value of the following form (lines 21 to 23 in Listing 3.4): As attribute list, we choose the attribute list from the table within the first argument query (*getAttrL (getTable q1)*). We produce the corresponding raw table by calling the auxiliary function *rawUnion* to produce the union of the raw tables within both argument queries (*getRaw (getTable q1)* and *getRaw (getTable q2)*). Obviously, this part of *reduce* only works out as expected if the two argument queries of the *union* query actually refer to tables with the same attribute list.

If the first table is a table value, but the second is not, we reduce the second argument query *q2* one step further by calling *reduce*, producing an intermediate *union* query with the result of that step as result (lines 24 to 27 in Listing 3.4). If the first table is not a table value, we reduce the first argument query *q1* one step further and produce an intermediate *union* query with the result as result (lines 28 to 31 in Listing 3.4).

```
1  primrec reduce :: ('id, 'val) Query ⇒ ('id, ('id, 'val) Table) Env ⇒
2     (('id, 'val) Query) option
3  where
4  reduce (tvalue t) ts = None |
5  reduce (selectFromWhere s tnl p) ts =
6    (if (length tnl = 1)
7    then (let n = hd tnl in
8      (let maybeTable = lookupEnv n ts in
9        (case maybeTable of
10         None ⇒ None |
11         Some t ⇒
12         (let filtered = filterTable t p in
13           let maybeSelected = projectTable s filtered in
14             (case maybeSelected of
15               None ⇒ None |
16               Some tsel ⇒ Some (tvalue tsel))))))
17   else None) |
18 reduce (union q1 q2) ts =
19   (if (isValue q1)
20   then (if (isValue q2)
21         then Some (tvalue
22           (table (getAttrL (getTable q1))
23             (rawUnion (getRaw (getTable q1)) (getRaw (getTable q2)))))
24         else (let q2reduce = (reduce q2 ts) in
25         (case q2reduce of
26           None ⇒ None |
27           Some q ⇒ Some (union q1 q))))
28   else  (let q1reduce = (reduce q1 ts) in
29     (case q1reduce of
30     None ⇒ None |
31     Some q ⇒ Some (union q q2))))
```

Listing 3.4.: Excerpt of small-step reduction semantics of SQL in Isabelle

```
1 fun rawUnion :: 'val RawTable ⇒ 'val RawTable ⇒ 'val RawTable
2 where
3 rawUnion Nil rt2 = rt2 |
4 rawUnion rt1 Nil = rt1 |
5 rawUnion (r1#rt1) rt2 =
6   (let urt1rt2 = rawUnion rt1 rt2 in
7     (if (r1 ∈ (set rt2))
8       then urt1rt2
9       else (r1#urt1rt2)))
```

Listing 3.5.: Function for producing the union of two raw tables in Isabelle

The other two set cases are treated in a similar fashion, so we omit their presentation here.

Next, we look into some of the low-level auxiliary functions called by *reduce* to give an impression of the "low-level" details which one has to deal with in order to fully specify the semantics of our subset of SQL.

The union of two tables is still relatively simple: We specify a recursive function *rawUnion*, shown in Listing 3.5. Note that the pattern structure of *rawUnion* is a little more complicated than just pattern matching on a single function argument, hence we cannot use Isabelle's **primrec** construct anymore, but instead have to use **fun**. If one of the two argument raw tables is empty (*Nil*), we directly return the other argument (lines 3 and 4 of Listing 3.5). In fact, line 4 could be omitted, this case simply shortens the evaluation, allowing to produce a result directly in case the second raw table is empty instead of having to traverse the entire first raw table. If the given raw tables are not empty, we recursively call *rawUnion* for the tail of the the first argument table (line 6 in Listing 3.5), whose result (*urt1rt2*) then becomes part of the final result. In lines 7 to 9 in Listing 3.5, we omit duplicates that would occur due to the union: We prepend the first row *r1* of the first argument table to the result of the recursive call only if this row is not present in the second argument table.

If the two argument tables did not contain any duplicate rows previously, the result of *rawUnion* will also not contain any duplicates, which exactly corresponds to the behavior of UNION queries in SQL. If, however, the argument tables already contained duplicate rows, these will not necessarily be removed by *rawUnion*. This is a slight simplification with regard to the original SQL behavior, which we introduced to keep the specification a bit simpler here.

The specification of table projection is much more involved, since projection involves low-level manipulation of (raw) table rows. We sketch a subset of the specification to give a flavor of the complexity.

Listing 3.6 contains the top-level function for projection on table columns. Function *projectTable* distinguishes the case of simple projection on *all* columns of a table, where it simply returns the entire given table (line 3 in Listing 3.6), from the remaining cases (projection on a given list of attribute names). In the latter case, *projectTable* calls another auxiliary function, *projectCols* (line 5 in Listing 3.6).

```
1  primrec projectTable :: 'id Select ⇒ ('id, 'val) Table ⇒ (('id, 'val) Table) option
2  where
3  projectTable all t = Some t |
4  projectTable (list al) t =
5  (case (projectCols al (getAttrL t) (getRaw t)) of
6    None ⇒ None |
7    Some rt ⇒ Some (table al rt))
```

Listing 3.6.: Top-level function for column projection on tables in Isabelle

```
1  primrec projectCols :: 'id AttrL ⇒ 'id AttrL ⇒ 'val RawTable ⇒ ('val RawTable) option
2  where
3  projectCols Nil al rt = Some (projectEmptyCol rt) |
4  projectCols (a#alr) al rt =
5    (let col = (findCol a al rt) in
6      (let rest = (projectCols alr al rt) in
7       (case col of
8        None ⇒ None |
9        Some rt1 ⇒ (case rest of
10           None ⇒ None |
11           Some rt2 ⇒ Some (attachColToFrontRaw rt1 rt2)))))
```

Listing 3.7.: Auxiliary function for column projection on tables in Isabelle

We show function projectCols in Listing 3.7. This auxiliary function does the actual projection work. Function *projectCols* receives two attribute lists as argument. The first attribute list contains the list of column names for projection, the second one the list of attribute names from the table header of the raw table that is passed as third argument to *projectCols*. The table's argument list is not interesting for the function *projectCols* itself, it is passed on to another auxiliary function (line 5 in Listing 3.7). Function *projectCols* is recursive in its first argument: If the attribute list for projection is empty, the function simply returns an empty column (line 3 in Listing 3.7). Note that in order to produce a sensible result table, it is important that *projectCols* does not simply return an arbitrary empty table in this case, but a table with exactly as many empty rows as the argument *rt* has. This empty column is produced by the primitive recursive helper function *projectEmptyCol*, whose specification we omit here. If the attribute list for projection is not empty, *projectEmptyCol* proceeds as follows (lines 4 to 11 in Listing 3.7):

1. The function *projectCols* calls the helper function *findCol* (specification below) to locate a column with the name *a* in the given table header *al* and extract it from the given raw table *rt* (line 5). This step may fail if the column *a* is not present in *al*. If it fails, *projectCols* also fails (returning *None*).

2. The function *projectCols* recursively calls itself with the tail of the attribute list for projection *alr* (line 6 Listing 3.7). This step may fail if one of the column names in *alr* cannot be found. If it fails, *projectCols* also fails (returning *None*).

```
1 fun findCol :: 'id ⇒ 'id AttrL ⇒ 'val RawTable ⇒ ('val RawTable) option
2 where
3 findCol n Nil rt = None |
4 findCol n (a#al) rt = (if (n = a)
5                       then (Some (projectFirstRaw rt))
6                       else (findCol n al (dropFirstColRaw rt)))
```

Listing 3.8.: Locating a single column within a table in Isabelle

3. If both of the first steps successfully returned a raw table as result, the function *projectCols* returns the result of the helper function *attachColToFrontRaw* (specification omitted here), which prepends the single-element rows within *rt1* to the rows within *rt2* (line 11 in Listing 3.7). Note that *attachColToFrontRaw* is specified so that it cannot fail, but the raw table it returns will only be sensible if indeed the rows within *rt1* all only contain one element and if the number of rows within *rt1* and *rt2* is the same.

The helper function *findCol* (see Listing 3.8) has a primitive recursive structure. Nevertheless, we specify it as **fun** in Isabelle instead of **primrec**, since later in the proofs, we need to access a specific induction rule for this function which the **primrec** construct does not give us. Function *findCol* recursively traverses the given attribute list of the given raw table *rt*. If the head of the table's attribute list *a* at one point equals the attribute name *n* we are looking for, we retrieve the first column of *rt* by calling the helper function *projectFirstRaw* (specification omitted here) in line 5 of Listing 3.8. For the recursive call of *findCol* in line 6 of Listing 3.8, we have to make sure that the column size of the argument raw table *rt* matches the shortened table attribute list. Hence, we call the helper function *dropFirstColRaw* (specification omitted here) to drop the first column of a given raw table. Both the helper functions *projectFirstRaw* and *dropFirstColRaw* are specified so that they cannot fail and will preserve the row count as well as the row lengths of the given raw table. Note that for simplicity, *findCol* will ignore any duplicate column names that might be present in the given attribute header, simply always returning the first column with name *n* that it finds along the way.

We briefly sketch how we specify row selection, omitting details of the specification. Overall, row selection is a little easier than column projection since we decided to specify tables as lists of rows instead of as lists of columns. Specifying tables as lists of rows is easier for constructing and extending tables with additional entries, but makes column projection harder. Basically, for row selection, we recursively traverse the rows of the given raw table and evaluate the predicate given in the SELECT FROM WHERE query for every row. Evaluating the predicates for row selection involves calling the underspecified functions *lt* and *gt* from locale *Table* (see Listing 3.1). If the given selection predicate evaluates to true for a given row, we include that row in the intermediate result table constructed during row selection, otherwise we omit it from the result.

```
1  type-synonym ('id, 'ftype) TType = ('id × 'ftype) list
2
3  locale FieldTypes =
4  fixes fieldType :: 'val ⇒ 'ftype
5
6
7  locale TypedTables = Table lt gt + FieldTypes ft
8  for lt :: 'val ⇒ 'val ⇒ bool
9  and gt :: 'val ⇒ 'val ⇒ bool
10 and ft :: 'val ⇒ 'ftype
11 begin
```

Listing 3.9.: Basic data structures for table types in Isabelle

Table Types

The semantics for our subset of SQL as we specified it has several points where it attempts "blindly" to construct a result table, without checking whether it is "sensible" to compose tables as required by a specific table. For example, it will blindly attempt the union of two tables even if both tables have different table headers. While this behavior will not clutter the semantics with checks that are unnecessary for "sensible" queries, it may result in queries failing at run time or in producing "insensible" tables.

We formally specify a type system for queries from our subset of SQL, The type system will check prior to the reduction of queries whether a query is "sensible" and will only produce "sensible" tables.

We first define basic data structures for typing queries in Listing 3.9. We decide that the type of a query is the type of the table this query produces. The type of a table is a typed table header, i.e. a list of pairs of attribute names and types of table fields (column types) (line 1 in Listing 3.9). Again, we leave concrete field types underspecified and instead use a type parameter *'ftype*. Next, we add an underspecified function *fieldType*, which defines the concrete field type of a table value, by adding it as a locale parameter (lines 3 and 4 of Listing 3.9). We combine the locale parameters from locale *Table* (see Listing 3.1) and locale *FieldTypes* within locale *TypedTables* to obtain a common context for our proofs (lines 7 to 11 in Listing 3.9). Note that for making sure that the signatures of all locale parameters have the desired form (and no undesired renaming of type parameters takes place) we have to restate the signatures of all locale parameters when defining the combined locale. We develop the rest of the specification of the type system as well as the proofs within this common context.

Next, we define a number of predicates the specify what it means for tables to be well-typed in Listing 3.10 with regard to a given table type. To be well-typed, firstly the attribute names within the given table type and the header of the given table have to match exactly, checked by predicate *matchingAttrL* (line 1 to 4 in Listing 3.10). Secondly, every row within the given raw table has to be well-typed with regard to the given table type, checked by predicate *welltypedRow* (lines 6 to 9

```
1  definition matchingAttrL :: ('id, 'ftype) TType ⇒ 'id AttrL ⇒ bool
2  where
3  matchingAttrL tt al ≡ ((length tt) = (length al)) ∧
4    (map (λtt. fst tt) tt) = al
5
6  definition welltypedRow :: ('id, 'ftype) TType ⇒ 'val Row ⇒ bool
7  where
8  welltypedRow tt r ≡ ((length tt = (length r)) ∧
9    (map (λv. ft v) r) = (map (λtt. snd tt) tt))
10
11 definition welltypedRawtable :: ('id, 'ftype) TType ⇒ 'val RawTable ⇒ bool
12 where
13 welltypedRawtable tt rt ≡ filter (λr. welltypedRow tt r) rt = rt
14
15 definition welltypedtable :: ('id, 'ftype) TType ⇒ ('id, 'val) Table ⇒ bool
16 where
17 welltypedtable tt t ≡ matchingAttrL tt (getAttrL t) ∧
18   welltypedRawtable tt (getRaw t)
```

Listing 3.10.: Predicates for well-typed tables in Isabelle

in Listing 3.10): The row's length has to be equal to the length of the table type, and the type of each table value within the row (looked up via the underspecified function *ft*) has to match the corresponding field type within the given table type. The top-level predicate *welltypedtable* (lines 15 to 18 in Listing 3.10) combines these conditions within a single predicate.

Type System

Finally, we specify the top-level type system of our subset of SQL as an inductive predicate within Isabelle, shown in Listing 3.11. We first define a typing judgment with three arguments (lines 1 to 3 in Listing 3.11): 1) a context that stores table types for table names, using the parametric data structure for environments that we defined in Listing 3.2, 2) a query, and 3) a table type (the type of the given query in the given table type context). Isabelle allows us to introduce syntactic sugar for inductive predicates to mimic the typical mathematical notation for typing judgments from the literature ($- \vdash - : -$).

Lines 5 to 18 of predicate *typable* define the individual named typing rules of our subset of SQL. Rule *Ttvalue* (line 5 in Listing 3.11) defines well-typedness of table values simply via predicate *welltypedtable*. Rule *TSelectFromWhere* (lines 6 to 9 in Listing 3.11) types SELECT FROM WHERE queries: The first premise checks whether the given list of table names *al* only contains one table name *tn*, since we decided to only consider selections and projections on single tables. The second premise looks the table type of *tn* up in the table type context *TTC*. The third premise (line 7) checks the row selection predicate *p* against the looked up table type *TT* of table *tn*, using a helper predicate *tcheckPred*, whose specification we omit

```
1  inductive typable :: ('id, ('id, 'ftype) TType) Env ⇒
2    ('id, 'val) Query ⇒ ('id, 'ftype) TType ⇒ bool
3  (- ⊢ - : -)
4  where
5  Ttvalue : (welltypedtable TT (table al rt)) ⟹ TTC ⊢ (tvalue (table al rt)) : TT |
6  TSelectFromWhere : ⟦ al = tn#Nil; lookupEnv tn TTC = Some TT;
7                       tcheckPred p TT;
8                       projectType s TT = Some TTr ⟧ ⟹
9                     TTC ⊢ (selectFromWhere s al p) : TTr |
10 TUnion : ⟦ TTC ⊢ q1 : TT;
11           TTC ⊢ q2 : TT ⟧ ⟹
12         TTC ⊢ (union q1 q2) : TT |
13 TIntersection : ⟦ TTC ⊢ q1 : TT;
14                   TTC ⊢ q2 : TT ⟧ ⟹
15         TTC ⊢ (intersection q1 q2) : TT |
16 TDifference : ⟦ TTC ⊢ q1 : TT;
17                 TTC ⊢ q2 : TT ⟧ ⟹
18         TTC ⊢ (difference q1 q2) : TT
```

Listing 3.11.: Type system for subset of SQL in Isabelle

here. Predicate *tcheckPred* checks whether all attribute names appearing within p are present as attribute names within table type TT. The fourth premise of rule *TSelectFromWhere* (line 8) checks whether the helper function *projectType* produces a result table type TTr, which ultimately becomes the result type of the given SELECT FROM WHERE query in the conclusion of the rule. We omit the full Isabelle specification of *projectType* here. The helper function *projectType* extracts all attribute names referred to within s from table type TT. Regarding potential duplicate attributes, we make sure that *projectType* operates like *projectTable*: It always simply retrieves the first (leftmost) attribute name within a given table type.

3.2.2. Progress and Preservation Proof of SQL

Having now specified the small-step reduction semantics and type system of our subset of typed SQL in Isabelle, we can proceed with stating and proving a progress and a preservation theorem. For both theorems, we need an additional premise which relate a table store and a table type context with each other. For stating this premise, we define a recursive predicate *StoreContextConsistent*, shown in Listing 3.12. This consistency predicate only returns true if both the given table store and the given table type context have the same length, if the attribute names and their order within both environments is the same, and if all tables in the table store are well-typed with regard to the table type with the same name at the same position in the table type context.

We use Isabelle's Isar syntax [Wen02] to state theorems and lemmas and to conduct proofs. The Isar syntax is relatively verbose, but close to how one would state proofs on paper and therefore very suitable for presentation. The theory file

```
1 fun StoreContextConsistent :: ('id, ('id, 'val) Table) Env ⇒
2   ('id, ('id, 'ftype) TType) Env ⇒ bool
3 where
4 StoreContextConsistent empty empty = True |
5 StoreContextConsistent (bind tn1 t tsr) (bind tn2 tt ttcr) =
6   ((tn1 = tn2) ∧ welltypedtable tt t ∧ StoreContextConsistent tsr ttcr) |
7 StoreContextConsistent ts ttc = False
```

Listing 3.12.: Predicate for relating table stores and table type contexts in Isabelle

for the progress proof contains about 370 lines of Isabelle code, the theory file for the preservation proof about 760 lines of Isabelle code. Both files contain almost all auxiliary lemmas needed for these two proofs, with the exception of some smaller inversion lemmas which are included in the theory files containing the specification of our typed subset of Isabelle. The preservation proof also uses some lemmas from the progress proof.

We show and explain interesting excerpts from the top-level progress and preservation proof in Isabelle/HOL to give an impression of how the final, full soundness proof of our example DSL looks like in Isabelle. In the following section, we will report how we obtained this proof within Isabelle.

Progress Proof

Listing 3.13 shows the progress theorem in Isabelle together with the first lines of the Isar proof script (lines 6 and 7). We state three named premises in lines 2, 3, and 4, respectively. Line 5 of Listing 3.13 contains the conclusion of the progress theorem. Intuitively, the theorem states that

- if an arbitrary given query q is not a table value already (premise *noValue*),
- if q is typable with a table type TT in a table type context TTC (premise *typable*),
- and if a table store TS is consistent with TTC (premise *consistency*),
- then q can be reduced to another query, taking TS as table store for the reduction.

In line 7 of Listing 3.13, we start an Isar proof by structural induction on q. Via line 6, we ensure that the premises from line 2 to 4 are taken into account when the induction cases are generated by the command in line 7. Next, we prove each of the 5 generated induction cases separately. We show the proof of the *selectFromWhere* case (Listing 3.14) and an excerpt of the proof of the *union* case (Listing 3.15).

For proving the *selectFromWhere* case, we first apply a case distinction on whether the length of the given list of table names *tnl* has length 1 or not (line 3 in Listing 3.14). The case for when the condition is *False* (line 20 to 23) is proven by contradiction: The second premise from the *selectFromWhere* case (named *selectFromWhere.prems(2)* within the Isar proof) gives us that the *selectFromWhere* query in this case is typable. Together with type inversion for this case (*selectFromWhere−INV*), it follows that the length of *tnl* has to be 1. This is a direct contradiction to the case assumption, so we

```
theorem SQLProgress:
  assumes noValue: ¬ (isValue  q)
  assumes typable: TTC ⊢ q : TT
  assumes consistency: StoreContextConsistent TS TTC
  shows ∃ q'. reduce q TS = Some q'
  using assms
proof (induction q)
```

Listing 3.13.: Progress theorem of typed SQL in Isabelle

```
  case (selectFromWhere s tnl p)
  then show ?case
  proof (cases length tnl = 1)
    case True
    obtain tt where lTT: lookupEnv (hd tnl) TTC = Some tt
      by (metis TypedTables.selectFromWhere-INV list.sel(1)
            selectFromWhere.prems(2))
    obtain t where ltable: lookupEnv (hd tnl) TS = Some t
      by (meson TypedTables.successfulLookup consistency lTT)
    have wtt: welltypedtable tt t
      by (meson TypedTables.welltypedLookup consistency lTT ltable)
    have wft: welltypedtable tt (filterTable t p)
      by (simp add: filterPreservesType wtt)
    obtain tsel where pft: projectTable s (filterTable t p) = Some tsel
      by (metis TypedTables.selectFromWhere-INV lTT list.sel(1)
           option.inject projectTableProgress selectFromWhere.prems(2) wft)
    show ?thesis
      by (simp add: True ltable pft)
  next
    case False
    thus ?thesis
      by (metis One-nat-def length-Cons list.size(3) selectFromWhere.prems(2)
          selectFromWhere-INV)
  qed
```

Listing 3.14.: Proof for SELECT FROM WHERE case from progress theorem in Isabelle

can directly prove the case in line 22 and 23 via an automatically generated proof script (more explanation on how to develop proofs follows in Subsection 3.2.3).

The case where the length of *tnl* is 1 (case *True*, from line 4 to 18 in Listing 3.14) requires a number of intermediate steps:

1. We obtain the table type *tt* which is the result of looking up the single table name in *tnl* in the given table type context *TTC* (lines 5 to 7). This table type exists since the *selectFromWhere* query is typable (*selectFromWhere.prems(2)*) and the premises of the typing rule for *selectFromWhere* queries gives us the condition in line 5.

2. Based on the previous step, we obtain the corresponding table *t* with table name *tnl* from the given table store *TS* (lines 8 to 10). We prove the existence of this table via the *consistency* premise from the progress theorem and an auxiliary lemma *successfulLookup* (specification omitted here). Lemma *successfulLookup* states that if a table name can successfully be looked up in a table type context, then this name can also be successfully looked up in a table store that is consistent with the table type context.

3. We argue that table *t* is well-typed with table type *tt* (line 10 and 11), using the previous steps, the *consistency* premise of the theorem, and the auxiliary lemma *welltypedLookup* (specification omitted here). Lemma *welltypedLookup* states that if you look up the same table name in a table type context *and* in a table store that are consistent with each other, the resulting table is well-typed with regard to the resulting table type.

4. We argue that the result of applying row selection with predicate *p* to table *t* generates a table that is well-typed with table type *tt* (lines 12 and 13), using the previous fact and the auxiliary lemma *filterPreservesType* (specification omitted here). Lemma *filterPreservesType* states that if a table was well-typed prior to row selection with a table type *tt*, it is still well-typed with that type after row selection.

5. We argue that column projection on columns in *s* of table *t* after the row selection step successfully produces a result table *tsel* (lines 14 to 16), mainly using the previous step and the auxiliary lemma *projectTableProgress* (specification omitted here). Lemma *projectTableProgress* states that column projection on a well-typed table (via the auxiliary function *projectTable*) is successful.

Finally, we use the fact that *tsel* exists to prove the conclusion of the *selectFromWhere* case of progress, i.e. that a well-typed *selectFromWhere* query can always take a step via the reduction semantics.

Proving the progress of *union* queries proceeds via case distinction on whether its argument queries *q1* and *q2* are table values or not, following the structure for this case within the reduction semantics (Listing 3.4). If both queries are table values already (lines 7 to 9 in Listing 3.15), progress of the union case follows directly by the definition of *reduce*, which cannot fail in this case.

```
1   case (union q1 q2)
2   then show ?case
3     proof (cases isValue q1)
4       case q1t: True
5       then show ?thesis
6         proof (cases isValue q2)
7           case q2t: True
8           with q1t show ?thesis
9             by simp
10        next
11          case q2f: False
12          have tq2: TTC ⊢ q2 : TT
13            using TypedTables.union-INV union.prems(2)
14            by blast
15          obtain q2' where rq2: reduce q2 TS = Some q2'
16            using consistency q2f tq2 union.IH(2)
17            by blast
18          obtain t where tv: q1 = tvalue t
19            using isValue-true-INV q1t
20            by blast
21          with rq2 q2f show ?thesis
22            using q1t
23            by auto
24        qed
25      next
26      case False
27  ...
28    qed
```

Listing 3.15.: Excerpt of proof for UNION case from progress theorem in Isabelle

If the second argument query is not a table value (lines 11 to 23 in Listing 3.15, we essentially only have to apply the induction hypothesis for *q2* (*union.IH(2)*) to obtain the result of the reduction step of *q2*. The application of the induction hypothesis takes place in lines 15 to 17. Then, we can use the result of the step of *q2* to show that in the present case, the top-level *union* query can be reduced one step further (lines 21 to 23). These two main steps require two auxiliary ones in Isabelle: Firstly, to argue that we can indeed apply the induction hypothesis for *q2* here, we need to show that *q2* is typable (lines 12 to 14, via inversion of the typing rule for *union*). Secondly, we need to argue about the concrete shape of the first argument query, since the reduction semantics does not use the *isValue* predicate, but pattern-matches on the shape of the queries. This argument is given in lines 18 to 20, via inversion of the *isValue* function (an auxiliary lemma we have to specify).

Proving the case where the first argument query is not a table value works essentially analogous to proving the previous case, hence we omitted the corresponding Isar script from Listing 3.15 (8 lines of Isar code).

```
1 theorem SQLPreservation:
2 assumes typable: TTC ⊢ q : TT
3 assumes step: reduce q TS = Some q'
4 assumes consistency: StoreContextConsistent TS TTC
5 shows TTC ⊢ q' : TT
6   using assms
7 proof (induction q arbitrary: q')
```

Listing 3.16.: Preservation theorem of typed SQL in Isabelle

Overall, the proof of the progress theorem for our subset of typed SQL in Isabelle requires stating and proving 12 auxiliary lemmas.

Preservation Proof

We show the preservation theorem and the top-level proof commands in Isabelle in Listing 3.17. Lines 2 to 4 contain the premises of the theorem: We assume a typable query q (premise *typable*) which reduces at least one step further to another query q' (premise *step*). The table type context used during typing and the table store used during reduction have to be consistent with each other, just as in the progress theorem shown earlier (premise *consistency*). The conclusion (line 5 in Listing 3.17) we want to prove states that query q' is typable with the same table type TT as the original query q.

In line 7 of Listing 3.4, we start a proof by structural induction over query q. Again, like in the progress proof, we instruct Isabelle to take the theorem's assumptions into account when generating the induction cases. But for the preservation proof, it is in addition important to manually instruct Isabelle to universally quantify variable q' again in the generated induction hypotheses (via *arbitrary* q'). Without this command, Isabelle only universally quantifies each variable within the theorem once at the outside of each induction case. While for variables *TTC*, *TT*, and *TS* this is what we want, we have to allow that the values for q' in the induction hypotheses can differ for the values for q' in the case assumptions. Otherwise, our induction hypotheses would be too weak. We make this explanation more concrete when explaining the *union* case of the preservation proof.

Like for the progress proof, we show and explain here the proofs for the cases of SELECT FROM WHERE queries (*selectFromWhere*) and for UNION queries (*union*).

The proof of the *selectFromWhere* case is shown in Listing 3.17. First, just like in the progress proof, we need to apply a case distinction on whether the length of the given list of table names is 1 or not (high-level structure given by lines 4, 5, and 39 in Listing 3.17). The case for when the length of *tnl* is not 1 (line 39 to 42) is again proved by contradiction, like in the progress proof.

The case for when the length of *tnl* is 1 is the interesting case (lines 5 to 37). We describe the steps of that case from a high-level point of view. In the first part of the proof, we prove the existence of a number of intermediate tables that are generated during the individual steps of a successful reduction of a *selectFromWhere*

query (lines 8 to 20). During these steps, we obtain the appropriate variables for these intermediate tables so that we can use them in the remainder of the proof. Next, we prove auxiliary equations about the result table *tsel* and the result type TT (lines 21 to 26). The key intermediate step at the end of the proof is to prove that the result table *tsel* is well-typed with regard to the result type TT (lines 32 to 34), which then directly proves the conclusion of the case by applying typing rule *Ttvalue* (lines 35 and 36). To prove that key intermediate step (line 32), we need to prove the well-typedness of some of the intermediate tables resulting from table lookup in the store and from row selection (lines 27 to 31). For proving all of these intermediate steps that we now mentioned, we require several auxiliary lemmas: Firstly, typing inversion for the *selectFromWhere* case (*selectFromWhere−INV*). Secondly, we reuse some of the auxiliary lemmas already used in the progress proof (*successfulLookup*, *welltypedLookup*, and *filterPreservesType*). Thirdly, we need a major new auxiliary lemma, namely *projectTableWelltypedWithSelectType*, which is used to prove the key intermediate step (line 33).

The high-level structure of the *union* case of the preservation theorem, shown in Listing 3.18, follows the high-level structure of the *union* case of the progress theorem (see Listing 3.15). As intermediate facts used in all sub-cases, we first obtain the premises of the typing rule for the *union* case via inversion (*union−INV*). Next, we have to argue about three cases:

1. Both argument queries are values. Here, we prove that the auxiliary function *rawUnion* preserves typing, via an auxiliary lemma. We omit the concrete proof script in Listing 3.18 (32 lines of Isar code).

2. The second argument query *q2* is not a value. In this case, we argue via the induction hypotheses for *q2* (*union.IH(2)*) that *q2* takes another step (lines 19 to 21) whose result is well-typed with the type of the *union* query TT (lines 22 to 24). Note that here, we actually have to use the extra universal quantification of q' in the induction hypothesis, instantiating this variable with the q' to which *q2* reduces. If the induction hypothesis were fixed to the same q' as the premises of the *union* case, we would not be able to prove the present sub-case. Finally, with these intermediate facts, we prove that the result query in this case is well-typed (lines 24 and 25).

3. The first argument query *q1* is not a value. The proof of this case is analogous to the proof of the previous case, hence we omit the proof script in Listing 3.18 (10 lines of Isar code).

Overall, the proof of the preservation theorem for our subset of typed SQL in Isabelle requires stating and proving 16 auxiliary lemmas. These 16 lemmas are in addition to the lemmas stated for the proof of progress, some of which the preservation proof reuses.

```
1  case (selectFromWhere s tnl p)
2  then show ?case
3    using assms
4  proof (cases length tnl = 1)
5    case True
6    then show ?thesis
7    proof −
8      obtain n where ai: tnl = n # Nil
9        using selectFromWhere(1) selectFromWhere-INV
10       by blast
11     obtain tt' where lnttc: lookupEnv n TTC = Some tt'
12       using ai selectFromWhere.prems(1) selectFromWhere-INV
13       by fastforce
14     obtain t where lnts: lookupEnv n TS = Some t
15       by (meson consistency lnttc successfulLookup)
16     obtain tsel where ts: q' = tvalue tsel
17       using reduce-INV-selectFromWhere selectFromWhere.prems(2)
18       by blast
19     obtain ft where ftp: filterTable t p = ft
20       by simp
21     have pt: projectTable s ft = Some tsel
22       using ai ftp lnts reduce-INV-selectFromWhere selectFromWhere.prems(2) ts
23       by fastforce
24     have ptt: projectType s tt' = Some TT
25       using ai lnttc selectFromWhere.prems(1) selectFromWhere-INV
26       by fastforce
27     have wtt: welltypedtable tt' t
28       by (meson consistency lnts lnttc welltypedLookup)
29     have wtft: welltypedtable tt' ft
30       using wtt filterPreservesType ftp
31       by blast
32     have wtsel: welltypedtable TT tsel
33       using projectTableWelltypedWithSelectType pt ptt wtft
34       by blast
35     show ?thesis
36       by (metis Table.exhaust Ttvalue ts wtsel)
37     qed
38   next
39   case False
40   show ?thesis
41     using False selectFromWhere.prems(2)
42     by auto
43 qed
```

Listing 3.17.: Proof for SELECT FROM WHERE case from preservation theorem in Isabelle

```
1   case (union q1 q2)
2   have wtq1: TTC ⊢ q1 : TT
3     using union.prems(1) union-INV
4     by blast
5   have wtq2: TTC ⊢ q2 : TT
6     using union.prems(1) union-INV
7     by blast
8   then show ?case
9   proof (cases isValue q1)
10     case valueq1t: True
11       then show ?thesis
12       proof (cases isValue q2)
13         case valueq2t: True
14 ...
15       next
16         case valueq2f: False
17         then show ?thesis
18           proof −
19             obtain q where redq2: reduce q2 TS = Some q and red: q' = union q1 q
20               using reduce-INV-union-valueq1t-valueq2f union.prems(2) valueq1t valueq2f
21               by blast
22             have wt: TTC ⊢ q : TT
23               by (simp add: consistency redq2 union.IH(2) wtq2)
24             show ?thesis
25               by (simp add: TUnion red wt wtq1)
26           qed
27       qed
28     next
29       case valueq1f: False
30 ...
31       qed
```

Listing 3.18.: Excerpt of proof for UNION case from preservation theorem in Isabelle

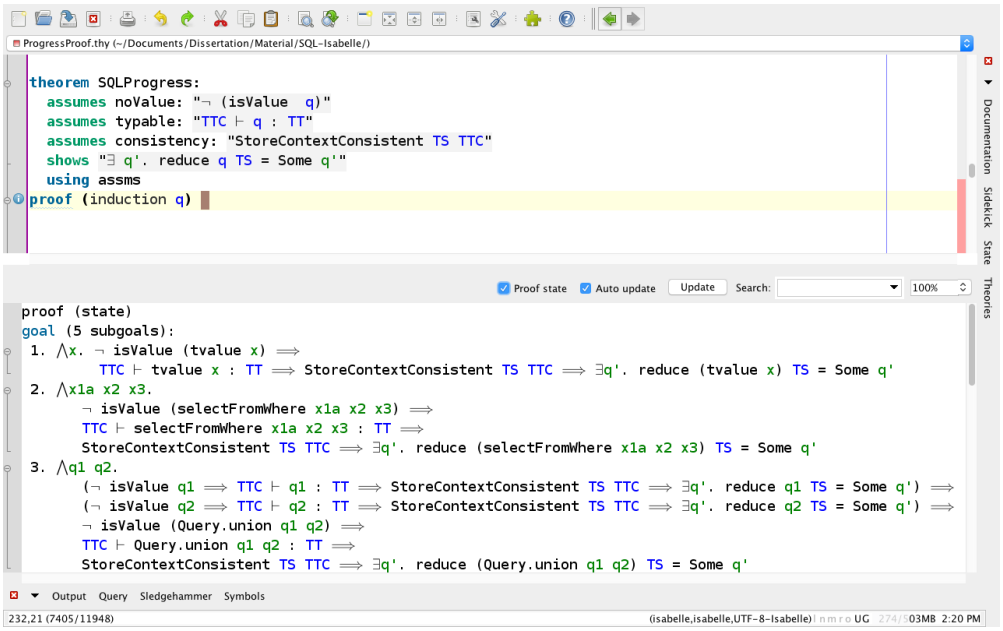


Figure 3.1.: Screenshot of top-level induction of the progress theorem for typed SQL within the Isabelle IDE

3.2.3. Discussion of the Proof Process

How did we obtain the proof that we presented in excerpts in the previous subsections? Which parts did we have to develop and type manually, and which parts could the existing automatic support within Isabelle generate? In this subsection, we answer these questions.

First, of course, the entire specification as well as the progress and preservation theorem had to be developed manually within Isabelle. Next, one has to manually figure out the top-level proof command (in our case, the structural induction on the queries). Figure 3.1 contains a screenshot of how this step looks like for the progress theorem within the Isabelle IDE. At the upper part of the screen, we see theorem *SQLProgress* and the top-level Isar command for induction we typed. At the lower level part of the screen, we see an excerpt of the resulting proof state, depicting the first three induction cases resulting from applying *induction q* (in total, there are 5 cases). The proof states within Isabelle are written within Isabelle's meta logic, where $\bigwedge$ represents universal quantification and $\Longrightarrow$ the implication arrow.

We now have to manually inspect the resulting induction cases and to decide whether these cases are indeed provable. For the progress proof, this is the case and we can proceed. However, as we explained earlier, applying only *induction q* at the beginning of the proof of the preservation theorem will generate some induction cases that cannot be proven, since the induction hypothesis is too weak. By inspecting the generated sub-cases, we have to figure this out by ourselves and modify the induction command accordingly (for the preservation theorem, as we explained in Subsection 3.2.2, we add *arbitrary: q'*).

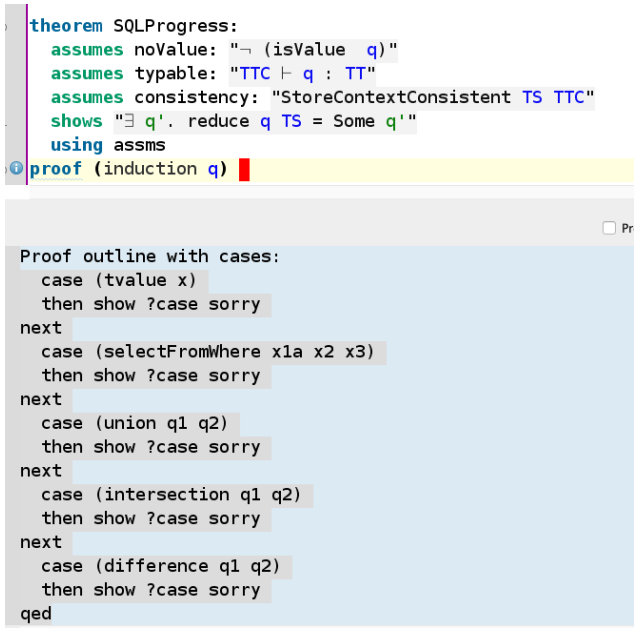


Figure 3.2.: Screenshot of suggested Isar proof structure for the top-level induction of the progress theorem for typed SQL within the Isabelle IDE

At the point where the cursor in Figure 3.1 is, Isabelle automatically suggests us a skeleton of an Isar proof to proceed below the proof state, shown in Figure 3.2. We can simply click on the suggested structure, which adds it automatically to the upper window of the Isabelle IDE (below the cursor) so that we can edit it and fill the gaps in the proof marked with `sorry`. The Isabelle IDE suggests high-level Isar proof structures for a number of top-level proof commands, e.g. also for case distinctions. This feature is especially convenient for Isabelle novices.

Next, the filling of the gaps within the suggested high-level proof structures is the part where Isabelle users have to invest manual work and creativity. One now has to come up with suitable intermediate facts to prove each case, and also with auxiliary lemmas wherever necessary (which will have to be proved as well). Intermediate facts of course have to be proved as well, by applying an appropriate proof tactic available within Isabelle. For this, the automation available within Isabelle is very helpful.

For example, in Figure 3.3, we show a screenshot of the development of the proof of the *selectFromWhere* case of the progress proof in the Isabelle IDE. We presented the corresponding final proof of this case in Subsection 3.2.2 in Listing 3.14. At the cursor position, we show how one would prove an intermediate step in a more "manual" fashion: Users would have to come up with which facts are necessary for proving this step by themselves, adding them via the Isar **from** command (here, facts *welltypedLookup* (auxiliary lemma), *consistency* (premise from progress theorem), *lTT*, and *ltable* (both facts previously developed within the proof of the case)). Then,

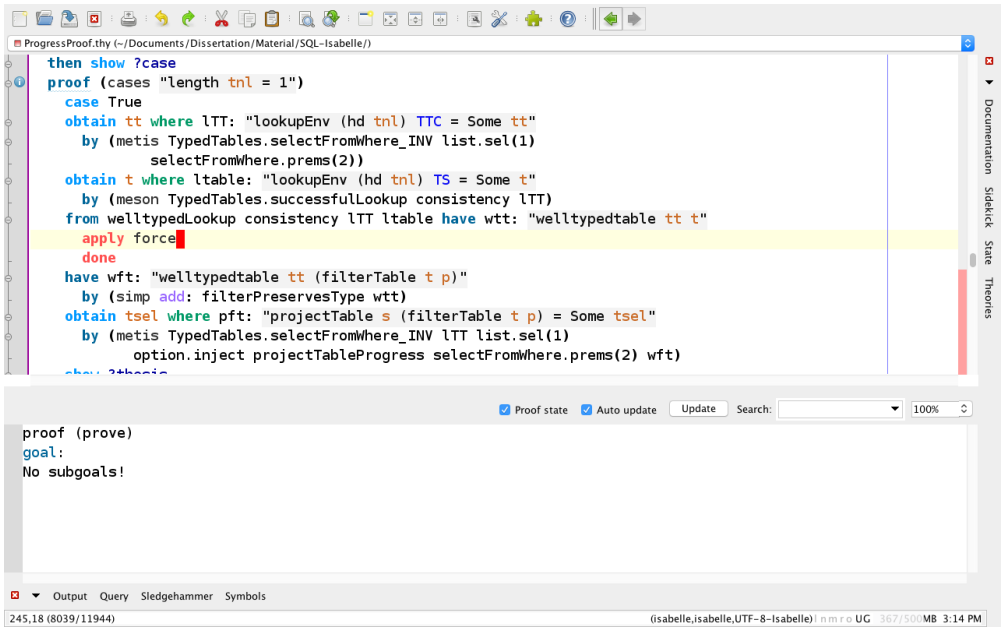


Figure 3.3.: Screenshot of intermediate state within the proof of the *selectFromWhere* case of the progress theorem for SQL in the Isabelle IDE.

one can apply one of Isabelle's automatic tactics to discharge the goal. Here, the tactic `force` is successful, producing "No subgoals!" within the proof state window, which means that the goal was completely proven. We close the apply-style proof via **done**. We can abbreviate these two lines by writing "**by** `force`", as we always did when presenting excerpts of the final proof script. This has the advantage that the overall proof script is much shorter. However, when inspecting the proof script within the Isabelle IDE, we can then not inspect intermediate proof states arising from tactic applications. For inspection, we would have to rewrite the proof scripts to the more elaborate version beforehand.

In Figure 3.4, we show how one can obtain the proof for the same step as in Figure 3.3 automatically within Isabelle: Instead of manually adding the necessary facts via **from**, we apply the `try` command, and wait a bit. Very often, one or more proof commands appear in the proof state window, often discharging the entire proof. In the lower part of Figure 3.4, we see that the "cvc4" prover has discovered a proof, using the facts *welltypedLookup*, *consistency*, *lTT*, and *ltable*. We can simply click on the discovered proof command, which copies it directly into our proof script in the upper window at the cursor position (and we may or may not erase the `try` command). (The proof commands below the cursor are marked red in Figure 3.4, since in that state, the proof at the cursor position is not closed, hence the remaining commands are not syntactically valid. They become valid as soon as the proposed proof is added.)

The `try` command calls various existing automatic tools within Isabelle, among them tools for generating counterexamples (Quickcheck [Bul12] and Nitpick [BN10a])

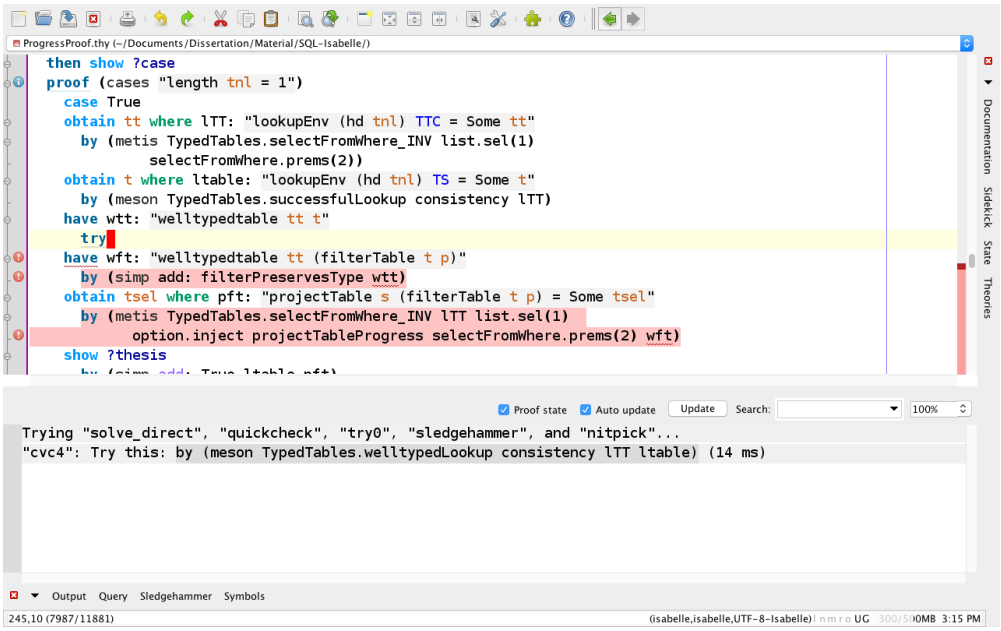


Figure 3.4.: Screenshot of intermediate state within the proof of the *selectFromWhere* case of the progress theorem for SQL in the Isabelle IDE. We use the `try` command to let Isabelle look for a proof for an intermediate step (cursor position).

as well as Sledgehammer [BBP11], which calls various automated theorem provers and SMT solvers with the proof problem at hand to see whether they return a proof. Isabelle reports any findings of the counterexample generators. Sledgehammer attempts to generate an appropriate proof command from any proofs reported by the external automated provers (a step which may also fail or generate proof commands that after all cannot be applied at the cursor position).

Within the progress and preservation proof of our subset of typed SQL, the `try` command has almost always been successful for finding proofs for the intermediate proof steps, even if one did not add any hints about which facts have to be used to discharge the proof. Only occasionally, one had to manually give a hint, adding one or more facts that are necessary for the proof. Whether this is necessary or not is not always clear and consistent, due to the nature of the heuristics used for the automated proof search. To distinguish within the proof when a fact was manually added and when it was discovered by applying `try`, we adopt the following convention within the Isar scripts we presented previously: If we added a fact before a **have** or **obtain** command (via **from**, **with**, or **thus**), it was manually added. Otherwise, if the fact's name appears in the proof after a **have**, **obtain**, or **show** command, it was discovered by `try`.

As an example of when manual interaction was necessary, let us reconsider the two proof excerpts of the *selectFromWhere* cases from the progress and from the preservation proofs. In the proof in Listing 3.14 (progress), lines 21 to 23, we need

to manually add the case assumption *False* to the proof (via the **thus** command in line 21) in order to obtain a successful proof command by calling `try`. However, in the analogous case in Listing 3.17 (preservation), `try` discovers the proof command without this manual addition (lines 40 to 42).

Over all, the Isabelle theory files for proving progress and preservation for our type system of SQL contain together about 1110 lines of uncommented Isabelle/Isar code. Roughly every second line of Isar code is automatically derived by Sledgehammer.

3.3. Using Dafny for Type Soundness Proofs

We now specify the same subset of typed SQL as in Section 3.2 in another verification system, namely in Dafny [Lei10]. Dafny was originally not designed for proofs of language specifications, but is powerful enough so that it is possible to specify type systems within Dafny and use the system for progress and preservation proofs. We specify our subset of typed SQL as close as possible to how we specified it in Isabelle. We show excerpts of the specification and of the proofs of progress and preservation which roughly correspond to the excerpts we showed and explained in Section 3.2. The full specification of our subset of typed SQL in Dafny as well as the progress and preservation proofs are available online (`https://bitbucket.org/cygne_noir/sql-dafny/src/master/`). However, in the present section we omit detailed explanations of the code, and refer the reader to the previous section for more details.

Again, the last subsection of this section discusses the proof process in Dafny (Subsection 3.3.3). Readers for whom the details of how we specify our typed subset of SQL in Dafny are not relevant at the moment may safely skip ahead to Subsection 3.3.3.

3.3.1. Specifying SQL Semantics and Type System

The structure of this subsection follows the structure of Subsection 3.2.1.

Modeling Tables

In Listing 3.19, we show how we model tables in Dafny. Like in Isabelle, we use lists of rows (which are lists of values) to model raw tables (which are lists of rows), parametric in a type for table values Val (lines 9 and 10). We declare two underspecified predicates greaterThan and lessThan for comparing two concrete table values of type Val (lines 4 and 6).

```
1  // attributes and field values and types
2  type AttrL<N> = seq<N>
3
4  predicate greaterThan<Val>(v1: Val, v2: Val)
5
6  predicate lessThan<Val>(v1: Val, v2: Val)
7
```

```
8   // "raw" table without header
9   type Row<Val> = seq<Val>
10  type RawTable<Val> = seq<Row<Val>>
11
12  // full table with header (attribute list)
13  datatype Table<N, Val> = table(getAL: AttrL<N>, getRaw: RawTable<Val>)
```

Listing 3.19: Basic data structures for tables in Dafny

We model parametric environments for storing named tables and, later, named table types in Listing 3.20. In lines 5 to 14, we see how the function for looking up values in environments looks like in Dafny.

```
1   datatype Option<A> = none | some(get: A)
2
3   datatype Env<N, A> = empty | bind(name: N, elem: A, rest: Env<N, A>)
4
5   function lookupEnv<N, A>(name: N, env: Env<N, A>): Option<A>
6   {
7     match env
8     {
9       case empty => none
10      case bind(m, a, e) =>
11        if name == m
12        then some(a)
13        else lookupEnv(name, e)
14    }
15  }
```

Listing 3.20: Environments for table/table type stores in Dafny

SQL Syntax

We define various datatypes for modeling SQL queries and parts of queries in Listing 3.21. The names of the datatypes and constructors and their intended meaning are like in our Isabelle specification of SQL from the previous section.

```
1   datatype Select<N> = all | list(getList: AttrL<N>)
2
3   type TRef<N> = seq<N>
4
5   datatype Exp<N, Val> = constant(Val) | lookup(N)
6
7   datatype Pred<N, Val> = ptrue
8     | and(Pred, Pred)
9     | not(Pred)
10    | eq(Exp<N, Val>, Exp<N, Val>)
11    | gt(Exp<N, Val>, Exp<N, Val>)
12    | lt(Exp<N, Val>, Exp<N, Val>)
13
14  datatype Query<N, Val> = tvalue(getTable: Table<N, Val>)
```

```
15      | selectFromWhere(Select<N>, TRef<N>, Pred<N, Val>)
16      | union(Query<N, Val>, Query<N, Val>)
17      | intersection(Query<N, Val>, Query<N, Val>)
18      | difference(Query<N, Val>, Query<N, Val>)
19
20    predicate isValue<N, Val>(q: Query<N, Val>)
21    {
22      q.tvalue?
23    }
```

Listing 3.21: Data structures for modeling SQL queries in Dafny

SQL Semantics

We define the small-step reduction semantics of our subset of SQL in Dafny via a recursive function, just like in the Isabelle specification. We show an excerpt of the analogous specification in Listing 3.22: Lines 9 to 16 specify the semantics of selectFromWhere questions, lines 17 to 23 specify how union queries are reduced, recursively calling reduce for reducing the argument queries of the union query. The rest of the cases are analogous to the union case, hence we omitted the specification of the remaining cases here.

```
1   type TStore<N, Val> = Env<N, Table<N, Val>>
2
3   function reduce<N, Val>(q: Query<N, Val>, ts: TStore<N, Val>):
4     Option<Query<N, Val>>
5   {
6     match q
7     {
8       case tvalue(t) => none
9       case selectFromWhere(sel, ref, pred) =>
10        if |ref| == 1
11        then if lookupEnv(ref[0], ts).some? &&
12              projectTable(sel, filterTable(lookupEnv(ref[0], ts).get, pred)).some?
13          then some(tvalue(projectTable(sel,
14                filterTable(lookupEnv(ref[0], ts).get, pred)).get))
15            else none
16        else none
17      case union(q1, q2) =>
18        if q1.tvalue? then
19          if q2.tvalue?
20          then some(tvalue(table(q1.getTable.getAL,
21                rawUnion(q1.getTable.getRaw, q2.getTable.getRaw))))
22          else if reduce(q2, ts).some? then some(union(q1, reduce(q2, ts).get)) else none
23        else if reduce(q1, ts).some? then some(union(reduce(q1, ts).get, q2)) else none
24      ...
25    }}
```

Listing 3.22: Excerpt of small-step reduction semantics of SQL in Dafny

We omit the specifications of the various auxiliary functions (e.g. filterTable) used within the reduction semantics. Their specifications are analogous to the ones

described and shown in the previous section.

Table Types

For modeling table types, we introduce a third type parameter into the specification, namely FType, abstracting over the concrete types of table values in fields. We introduce an underspecified function fieldType, which assigns types to table values - see line 1 in Listing 3.23. Finally, line 4 in Listing 3.23 shows how types of tables look like in the Dafny specification.

```
1  function fieldType<Val, FType>(v: Val): FType
2
3  //typed table schemas
4  type TType<N, FType> = seq<(N, FType)>
```

Listing 3.23: Basic data structures for table types in Dafny

Type System

We specify well-typedness of tables via predicates in Dafny, shown in Listing 3.24. The definition of the individual predicates is semantically equal to the definitions in Isabelle from Listing 3.10. The main abstract difference to the Isabelle specification is that we do not use higher-order functions on lists such as map and filter, but instead specify the predicates in a more "imperative" fashion. This works better together with Dafny's proof automation, which is heavily optimized for arguing about array and list accesses.

```
1  predicate matchingAttrL<N, FType>(tt: TType<N, FType>, al: AttrL<N>)
2  { |tt| == |al| && ∀ i :: 0 <= i < |tt| ==> tt[i].0 == al[i] }
3
4  predicate welltypedRow<N, FType, Val>(tt: TType<N, FType>, r: Row<Val>)
5  { |tt| == |r| && ∀ i :: 0 <= i < |tt| ==> fieldType(r[i]) == tt[i].1 }
6
7  predicate welltypedRawtable<N, FType, Val>(tt: TType<N, FType>, t:
       RawTable<Val>)
8  { ∀ i :: 0 <= i < |t| ==> welltypedRow(tt, t[i]) }
9
10 predicate welltypedtable<N, FType, Val>(tt: TType<N, FType>, t: Table<N, Val>)
11 { matchingAttrL(tt, t.getAL) && welltypedRawtable(tt, t.getRaw) }
```

Listing 3.24: Predicates for well-typed tables in Dafny

Finally, we specify the top-level type system of our typed subset of SQL. In Dafny, there is no construct for inductive definitions. However, predicates may be defined recursively. Since our specific type system is syntax-directed anyway, it is possible to formulate it in a more "algorithmic" fashion. We show the type system in Dafny in Listing 3.25. The individual cases within predicate typable can be seen to roughly correspond to the different typing rules from Listing 3.11. However, here, we have no named, inductively defined typing rules. Instead, we specify for each syntactic

case what has to be checked in order to determine whether the query in this case is well-typed or not (corresponding to the premises of the inductively defined rules in Listing 3.11). For the set queries (lines 9 to 11 in Listing 3.25), we recursively call predicate typable again.

```
1  predicate typable<N, Val, FType>(ttc: TTContext<N, FType>,
2    q: Query<N, Val>, tt: TType<N, FType>)
3  {
4    match q
5    {
6      case tvalue(t) => welltypedtable(tt, t)
7      case selectFromWhere(sel, refs, p) => |refs| == 1 && lookupEnv(refs[0], ttc).some?
8        && tcheckPred(p, lookupEnv(refs[0], ttc).get) && projectType(sel,
             lookupEnv(refs[0], ttc).get) == some(tt)
9      case union(q1, q2) => typable(ttc, q1, tt) && typable(ttc, q2, tt)
10     case intersection(q1, q2) => typable(ttc, q1, tt) && typable(ttc, q2, tt)
11     case difference(q1, q2) => typable(ttc, q1, tt) && typable(ttc, q2, tt)
12   }
13 }
```

Listing 3.25: Type system for subset of SQL in Dafny

We omit the specifications of the auxiliary functions used within predicate typable, e.g. projectType. They are specified like the corresponding auxiliary functions within the Isabelle specification.

3.3.2. Progress and Preservation Proof of SQL

Just like in the Isabelle proof, we first define via a recursive predicate when a table store is consistent with a table type context, shown in Listing 3.26. This predicate is used within the premises of both the progress and the preservation theorems.

```
1  predicate StoreContextConsistent<N, Val, FType>(ts: TStore<N, Val>,
2    ttc: TTContext<N, FType>)
3  {
4    (ts.empty? && ttc.empty?)
5      ||
6      (ts.bind? && ttc.bind? && ts.name == ttc.name
7      && welltypedtable(ttc.elem, ts.elem) && StoreContextConsistent(ts.rest, ttc.rest))
8  }
```

Listing 3.26: Predicate for relating table stores and table type contexts in Dafny

Progress Proof

Listing 3.27 shows the progress theorem in Dafny as well as the entire top-level proof of the theorem. We omit the presentation of the auxiliary lemmas (here successfulLookup, welltypedLookup, filterPreserversType, and projectTableProgress), just like we did when presenting the Isabelle proof. The parts where there are gaps within the proof commands (e.g. in lines 9, and 21 to 23 after the arrows) are parts which

are proven automatically by Dafny. The proof starts with a structural induction on variable q (line 7).

```
1  lemma progress<N, FType, Val>(ttc: TTContext<N, FType>, ts: TStore<N, Val>,
2      q: Query<N, Val>)
3    requires ∃ tt :: typable(ttc, q, tt)
4    requires StoreContextConsistent(ts, ttc)
5    ensures isValue(q) || reduce(q, ts).some?
6  {
7    match q
8    {
9      case tvalue(t) =>
10     case selectFromWhere(sel, refs, p) =>
11       if |refs| == 1
12       {
13         successfulLookup(ttc, ts, refs[0]);
14         welltypedLookup(ttc, ts, refs[0]);
15         var t := lookupEnv(refs[0], ts).get;
16         var tt := lookupEnv(refs[0], ttc).get;
17         filterPreservesType(tt, t, p);
18         projectTableProgress(sel, tt, filterTable(t, p));
19       }
20       else {}
21     case union(q1, q2) =>
22     case intersection(q1, q2) =>
23     case difference(q1, q2) =>
24   }
25 }
```

Listing 3.27: Progress proof of typed SQL in Dafny

As we can see, the three set cases (line 21 to 23) are proven automatically, no further user action is required. For the selectFromWhere case (lines 10 to 20), we need to give some high-level hints: First, the top-level case distinction regarding the length of the list of table names, just like in the Isabelle proof. The contradictory case is proven automatically by Dafny (line 20). Within the interesting case, we need to give as hints the concrete instances of the auxiliary lemmas that are needed (lines 13 and 14 as well as 17 and 18). To be able to specify some of these instances, we need to obtain proof variables t (the table looked up from the store) and tt (the table type looked up from the context) in lines 15 and 16. This roughly corresponds to the **obtain** steps from Listing 3.14. The remainder of the proof (the concrete order in which the lemma instances have to be applied to the proof goal etc.) is again proven automatically by Dafny.

Overall, the progress proof in Dafny requires stating and proving 10 auxiliary lemmas, out of which 5 are proven fully automatically by Dafny, i.e. without requiring any user interaction.

Preservation Proof

We show the Dafny version of the preservation theorem as well as the start of its proof in Listing 3.28.

```
1 lemma preservation<N, FType, Val>(ttc: TTContext<N, FType>, ts: TStore<N, Val>,
2     q: Query<N, Val>, q': Query<N, Val>, tt: TType<N, FType>)
3     requires typable(ttc, q, tt)
4     requires reduce(q, ts) == some(q')
5     requires StoreContextConsistent(ts, ttc)
6     ensures typable(ttc, q', tt)
7   {
8     match q ...
9   }
```

Listing 3.28: Preservation theorem of typed SQL in Dafny

The first proof command starts a structural induction on q (line 8). Note that Dafny internally considers the correct induction scheme for the preservation theorem. Unlike for the Isabelle proof of preservation, an "additional tweaking" of the command for beginning an induction is not required.

The proof of the preservation theorem requires slightly more hints than the proof of the progress theorem in Dafny.

We show the proof commands for the selectFromWhere case in Listing 3.29. Like in the selectFromWhere case of the progress proof, we need to manually specify the case distinction regarding the length of the list of table names in the query (refs). Again, the contradictory case can be proven automatically by Dafny (line 11). For the then-case, we have to specify the concrete instances of the auxiliary lemmas needed (lines 4 and 5, lines 8 and 9), along with equations for intermediate proof variables (lines 6 and 7). The rest is done automatically by Dafny.

```
1   case selectFromWhere(sel, refs, p) =>
2     if |refs| == 1
3     {
4       successfulLookup(ttc, ts, refs[0]);
5       welltypedLookup(ttc, ts, refs[0]);
6       var t := lookupEnv(refs[0], ts).get;
7       var tt' := lookupEnv(refs[0], ttc).get;
8       filterPreservesType(tt', t, p);
9       projectTableWelltypedWithSelectType(sel, filterTable(t, p), tt');
10    }
11    else {}
```

Listing 3.29: Proof of SELECT FROM WHERE case of preservation theorem of typed SQL in Dafny

The proof of the union case, shown in Listing 3.30, firstly requires manually specifying the necessary case distinctions, regarding whether the argument queries are table values or not (lines 2 and 4). The two of the resulting sub-cases that require applying the induction hypotheses are proven automatically by Dafny (lines

7 and 10). The case where both argument queries are table values requires specifying the instance of the auxiliary lemma needed in this case (line 5).

```
1   case union(q1, q2) =>
2       if q1.tvalue?
3       {
4         if q2.tvalue?
5         {rawUnionPreservesWellTypedRaw(q1.getTable.getRaw, q2.getTable.getRaw, tt);}
6         else
7         {}
8       }
9       else
10      {}
```

Listing 3.30: Proof of UNION case of preservation theorem of typed SQL in Dafny

Overall, the proof of the preservation theorem in Dafny requires 16 additional auxiliary lemmas (to the ones reused from the progress proof), out of which 10 are proven fully automatically by Dafny.

3.3.3. Discussion of the Proof Process

We developed the progress and preservation proof of our subset of typed SQL in Dafny via a top-down refinement strategy: For every lemma, we first tried to let Dafny automatically prove the lemma. If the automatic proof failed, we manually developed proof hints for refining the proof, inspecting Dafny's error messages. We iterated this process until all error messages disappeared.

For example, for developing the proof of the progress theorem, we first see that a fully automatic proof only produces an error message. We know that the top-level proof requires structural induction, so we manually specify the induction structure needed. Thus, we achieve the intermediate state shown in Figure 3.5. We use the Emacs mode available for Dafny. As we can see, Dafny returns certain error messages (lower part of the screenshot in Figure 3.5). Basically, these error messages tell us that a certain post-condition "might not hold" and which post-condition it is (in our case, the conclusion of our theorem). The Emacs mode of Dafny also underlines the corresponding related locations (upper part of the screenshot in Figure 3.5). From the related location, we can infer that the conclusion of our progress theorem cannot be proven for the selectFromWhere case. However, since there are no further error messages for the remaining cases, we can already see that these cases may be proven automatically by Dafny. (The verifier reports all problems found and does not get stuck at the first problem.)

We now attempt to refine the proof for the selectFromWhere case: We add the case distinction on the length of ref (see Listing 3.27). This gives us the same error message as shown in Figure 3.5), but with a slightly refined location: The error points to the **then** case of the new case distinction, while the **else** case does not have any error message attached. This means that at least the latter case can be proved automatically. Next, we add the instances of auxiliary lemma applications from

```
  lemma progress<N, FType, Val>(ttc: TTContext<N, FType>, ts: TStore<N, Val>, q: Query<
N, Val>)
    requires StoreContextConsistent(ts, ttc)
    requires ∃ tt • typable(ttc, q, tt)
    ensures isValue(q) ∨ reduce(q, ts).some?
  {
    match q
    {
      case tvalue(t) ⇒
      case selectFromWhere(sel, refs, p) ⇒
      case union(q1, q2) ⇒
      case intersection(q1, q2) ⇒
      case difference(q1, q2) ⇒
    }
  }

}
 -:---  ProgressProof.dfy   Bot (125,42)  Git:master  (Dafny hs yas company FlyC:1/1)
-*- mode: compilation; default-directory: "~/Documents/Dissertation/Material/dafny/SQL-
Dafny/" -*-
Compilation started at Fri Nov 30 11:46:40

/Users/sylvia/Documents/Dissertation/Material/dafny/dafny /compile\:0 /nologo /Users/sy
lvia/Documents/Dissertation/Material/dafny/SQL-Dafny/ProgressProof.dfy
/Users/sylvia/Documents/Dissertation/Material/dafny/SQL-Dafny/ProgressProof.dfy(125,3):
 Error BP5003: A postcondition might not hold on this return path.
/Users/sylvia/Documents/Dissertation/Material/dafny/SQL-Dafny/ProgressProof.dfy(120,21)
: Related location: This is the postcondition that might not hold.
Execution trace:
    (0,0): anon0
    (0,0): anon7_Else
    (0,0): anon8_Then

Dafny program verifier finished with 14 verified, 1 error

Compilation exited abnormally with code 4 at Fri Nov 30 11:46:47
```

Figure 3.5.: Screenshot of top-level induction of the progress theorem for typed SQL within the Dafny (Emacs mode)

```
Dafny program verifier finished with 15 verified, 0 errors

Compilation finished at Fri Nov 30 12:09:13
```

Figure 3.6.: Screenshot of final success message for progress theorem for typed SQL within the Dafny (Emacs mode)

Listing 3.27, until we reach the state shown in Figure 3.6. Unfortunately, we keep seeing the same error message as shown in Figure 3.5 (with refined locations) until we have added all the missing lines from Listing 3.27. As long as the lines we add to the proof do not produce additional error messages, we know at least that the lemma instances we add are somehow applicable here. But apart from this, we have no means of inspecting the proof state further. That is, we have to creatively think through ourselves how the corresponding intermediate goals might look like and which information might be missing for the verifier.

Overall, whenever it is necessary to apply an auxiliary lemma within the proof, we had to pass the appropriate lemma instance as a hint to the Dafny verifier. If a lemma was only needed within a specific sub-case of the proof, it was also necessary to first manually give the necessary case distinction structure, and then only give the appropriate lemma instance within the specific sub-case of the proof (see for example the `union` case in Listing 3.27).

3.4. Discussion of Different Existing Systems

In Sections 3.2 and 3.3, we presented in detail how one can use Isabelle respectively Dafny to develop a soundness proof of a type system of a language that is within our scope according to Subsection 3.1.2. In each section, we focused on the respective system and did not compare the systems against each other or against other systems.

In the present section, we will compare Isabelle/HOL and Dafny to each other (Subsection 3.4.1) and discuss the pros and cons of using further systems for such soundness proofs (Subsection 3.4.2). In the discussion, we focus on the general usability of the systems for non-experts and on the degree of proof automation achievable.

3.4.1. Isabelle/HOL vs. Dafny

To put the present comparison of Isabelle and Dafny into perspective, we first report some subjective meta data on the two proof developments we presented in Sections 3.2 and 3.3. The author of this dissertation originally developed both specifications roughly in parallel, starting out with about three years of Isabelle experience and no prior experience in using Dafny.

The author of this dissertation first completed the development of the soundness proof in Dafny. As we have seen, this development ultimately “only” involved

specifying a rough proof structure and the necessary auxiliary lemmas, due to Dafny's impressive verification automation. The very first proof attempt was slightly longer than the one presented in Section 3.3: It contained and used more auxiliary lemmas. In parallel, the author created the same rough proof structure within the Isabelle specification, with gaps (writing **sorry** in the Isar scripts). After finishing the first Dafny proof, the author proceeded to filling the gaps within the Isabelle proof. Since Isabelle requires specifying the individual proof steps in a much more fine-grained way than Dafny for our proofs, this process took somewhat longer. However, the author could base the proof on the structure of the Dafny proof, already knowing that this proof structure would work out and thus potentially saving effort otherwise spent with following "wrong paths".

Interestingly, the fine granularity of the Isar scripts forced the author to put some more thoughts into the overall proof structure. In between it turned out that some of the auxiliary lemmas from the original Dafny proof could be simplified so that other auxiliary lemmas could be entirely dropped. The author then first tried out the new structure within the Dafny proof. When it worked out, she adapted the Isabelle proof structure in the same way. This adaptation ultimately saved quite some effort within the development of the Isabelle proof, since no time was spent for the development of the Isabelle proofs of the unnecessary auxiliary lemmas.

Overall, the development of the Dafny proof, including learning Dafny and simplifying the first proof took the author about 5 full work days. The entire development of the Isabelle proof, largely based on the Dafny proof structure and with prior Isabelle knowledge, took the author in total about 12 full work days.

Comparison of usability for proof development We first compare both systems focusing on the given support for proof development, ignoring any considerations about time and effort. For developing a totally unknown proof and when having little experience with formal verification, the author of this thesis sees a slight advantage for Isabelle over Dafny: Dafny does not provide any means for closely inspecting intermediate proof states or internally generated auxiliary facts. Thus, developing a proof in Dafny involves a lot of "guessing around". For the author herself, this overall "guessing" led to a successful first proof, which ultimately turned out to be more complicated than necessary. In other cases, the lack of intermediate proof information could entirely prevent the completion of a proof.

Isabelle, on the other hand, reports the detailed proof state at every point within a proof. Isabelle also allows for querying any facts that were internally generated from specifications. And finally, as we also demonstrated in Subsection 3.2.3, Isabelle suggests overall proof structures for many top-level proof commands. Overall, these features help to come up with the overall proof, and also with understanding a proof thoroughly.

Comparison of available automation If one looks at the available automation features, thus focusing on the overall time and effort needed for proof development, Dafny clearly emerges as the "winner": Even with no prior Dafny experience, the author of this thesis could develop the type soundness proof for SQL much faster

than the Isabelle proof. While Isabelle's automation via the **try** command is also very helpful to shorten the time for developing proofs, Dafny's automation is simply stronger, and also capable of automating more complicated steps (e.g. inductions). Overall, the author would judge that especially with some prior ideas on the overall structure of a proof, it is typically much faster to develop a proof in Dafny than in Isabelle.

Comparison of final proof scripts How well can you understand a final proof from reading the resulting proof scripts in Isabelle or Dafny? We argue that Isabelle's Isar scripts are much more understandable than final Dafny scripts: Isar proof steps are written out in a human-readable format, following natural language. Also, the fine granularity of Isabelle's Isar scripts allows one to easily understand the details of a proof, even if one has little experience with Isabelle. Dafny scripts, on the other hand, are harder to read: First, you have to understand Dafny's proof language, which is relatively far away from proofs written in natural language. Next, the omissions within the scripts of steps that Dafny can automatically prove also prevent pure readers with no knowledge about the proof from understanding what is going on.

However, Dafny also allows you to develop more detailed proof scripts, relying less on the verification automation. But developing more detailed scripts takes of course more time - and is relatively difficult, since, as said before, one cannot inspect intermediate proof states within Dafny.

Comparison of initial learning effort Judging the initial learning effort for Isabelle and Dafny, the author of this thesis argues from her experience that learning Dafny is much easier and faster than learning Isabelle: Dafny contains less specification and proof features than Isabelle. The author herself was in only a few day's time able to use Dafny for developing a full type soundness proof. Reaching the same level of expertise in Isabelle took the author much longer.

However, the author's prior knowledge in the area of machine-checked, interactive verification and about Isabelle in particular probably helped her to learn Dafny faster and to be able to use the system for a quick proof development. It may be much harder for a person lacking any knowledge in the area of computer-aided verification.

Summary Overall, when comparing Isabelle against Dafny, Dafny's strengths lie in being a lean system with powerful automation. This is especially useful for "quick and dirty" proof development, provided one has prior experience in formal verification and hence does not have to completely "guess around" when developing a proof. Isabelle's strengths lie in supporting proof development with detailed feedback on the proof step and on producing human-readable proofs with a high level of details.

Both Isabelle and Dafny require a certain level of expertise in formal verification so that one can efficiently develop soundness proofs. Both systems require a certain amount of manual work to conduct the kinds of proofs we focus on in this thesis.

3.4.2. Formalizing Type Soundness Proofs in Other Systems

We discuss a selection of other existing systems that one could also use for mechanizing soundness proofs of type systems of DSLs, focusing on the automation features offered by these systems. We did not complete ourselves any soundness proofs within these systems, but studied their proof automation features.

Coq Next to Isabelle, another big and widely known interactive theorem prover is Coq [Tea19]. Coq has lots of features that are very similar to what Isabelle offers: Specifications are written within a dependently-typed, functional programming language, one can prove theorems/lemmas by writing tactic scripts, and one may also inspect intermediate goal state and final proofs (given as "proof terms", i.e. as a dependently-typed program that represents the proof, roughly spoken).

In Coq, one conducts proofs via tactic scripts. There is no actively maintained and widely used declarative proof language such as Isabelle/Isar within Coq. There are various automatic proof tactics in Coq, similar to Isabelle's built-in automatic tactics *auto* or *simp*. Coq also has support for calling external provers for automatic verification of certain proof parts. However, this support is not as far developed as Isabelle's Sledgehammer. This is mainly due to the underlying logics: Isabelle uses classical logic, which easily fits in with many external automated theorem provers, which typically also use classical logic. Coq, on the other hand, is based on intuitionistic logic, which is not a direct fit for existing ATPs. Apart from that, Coq provides powerful tactic languages for custom proof automation. We will talk more about proof automation via tactic languages in the next subsection.

The initial learning effort for Coq is similar to the learning effort for Isabelle: The system has a lot of powerful features and proof tactics, which require a certain level of understanding to be used efficiently.

Overall, using Coq for mechanizing type soundness proofs is very similar to using Isabelle, regarding the level of detail in the proofs you need to give manually, and regarding the initial learning effort and overall skill one needs for using the system. The support for automation of individual proof steps and for suggesting proof structures probably is a little more advanced within Isabelle, from the author's subjective perspective.

Twelf We described the parts of the Twelf system [PS99] already in Subsections 1.1.1 and 3.1.1, when discussing the "name-binding problem". To summarize the important points mentioned there in a different context: The Twelf system is designed for using a logical framework (LF) [Pfe91] for theorem proving, building on the HOAS approach for encoding name-binding. Thus, it is very well-suited for elegantly dealing with the "name-binding problem", but rather difficult to use for certain other aspects of proofs. Using Twelf requires special knowledge in the area of logical frameworks and theorem proving.

Regarding proof automation, Carsten Schürmann and Frank Pfenning designed a meta-logic for inductive reasoning over LF encodings, which they successfully used to automatically prove type preservation of Mini-ML [SP98]. "Automatically" here

means that any auxiliary lemmas that are needed have to be specified beforehand and also applied manually at the appropriate places - using LF-reasoning style. So again, also the "automatic" mode of Twelf requires quite some manual intervention and skill for the proofs we are focusing on in this thesis.

Why3 Why3 is a general-purpose verification platform for deductive program verification [Bob+11; FP13]. One specifies proof problems for Why3 via a rich specification and programming language called WhyML, an ML-like language supporting the specification of predicates (also inductive ones) and lemmas. External theorem provers, automated as well as interactive, discharge verification conditions. One triggers verification of a property in Why3 via specifying *proof tasks*, which translate a certain proof problem for an external prover to be solved automatically, reporting the output of the prover back. Users have the possibility to manually manipulate the translation of proof problems by applying certain "transformation" commands, which may be necessary to obtain more complex proofs. Also, Why3 allows for annotating WhyML programs with "ghost code", i.e. commands that facilitate verification, such as invariants of user-specified data structures.

Apart from calling different automated provers, Why3 does not offer as of yet any other language constructs for high-level automation. That is, users have to define themselves any auxiliary lemmas and complex intermediate steps in proofs. In Chapter 9, we will discuss in more detail how Why3 compares to our overall work in this thesis.

3.4.3. Full Proof Automation in Existing Systems?

Now we discuss how one could raise the degree of automation within existing systems in order to automate the proofs we are interested in as far as possible. Most of the existing systems we mentioned and discussed in this chapter feature so-called *tactic languages* for custom proof automation. A tactic language is a special-purpose script language for defining tactics within a verification system. Tactics transform proof goals to one or more intermediate proof goals, until ultimately completely proving a goal, if possible.

Coq features an established, untyped tactic language called LTac [Del00]. LTac expressions can only be applied in the context of a proof; their evaluation yields either a term or an integer (to be used as intermediate results) or a tactic (intended as the final result of any tactic expression). The final result of a tactic expression is applied to the focused goals to transform it to one or more subgoals. Tactic expressions may match on goals and intermediate terms, select certain goals, and apply existing as well as custom tactics to them. Ltac also features various combinators for tactic expressions, e.g. for backtracking, conditional application of tactics, etc. More recently, there is a second, typed tactic language for Coq called MTac [Zil+13] which supports dependently-typed tactic programming and provides a better integration with existing commands for Coq programming than LTac.

Isabelle offers a still relatively young tactic language called Eisbach [MMW16], inspired by LTac. The authors of the Eisbach language describe the language as

"proof method language": Eisbach provides a **method** command for defining new proof methods using the existing Isar syntax and combining pre-defined proof methods to new ones. Like LTac, Eisbach is an untyped language. Grov et al. developed a graphical representation for proof strategies, called PSGraphs, implemented within Tinker tool [GKL13]. In this representation, tactics appear on nodes in a graph and are connected by "piping" them together. Tinker tool works together with Isabelle and 2 other interactive theorem provers.

For Dafny, there is also a still relatively young tactic language called Tacny [GT16], inspired also by tactic languages for Coq and Isabelle. With Tacny, Dafny users can encode high-level proof patterns using a slightly extended Dafny syntax.

All of the tactic languages we mentioned have in common that they provide what one could call an "activity-oriented view" on a proof: To program a tactic (or proof method) via a tactic language, one has to come up with a series of *steps* that have to be applied to transform a goal, abstractly thinking about possible intermediate states within a proof. To successfully write or modify a tactic script, one also needs to be fluent with the existing tactics within a prover. Another common point is that a tactic script typically either is fully successful, or it fails at some intermediate step in a case that the tactic does not consider yet, which then makes the entire tactic fail. Debugging a failing tactic is typically hard and requires skillful user interaction. For LTac, there is an established debugger for tactic scripts which helps with discovering and fixing problems, but for the younger tactic languages such as Eisbach, the implementation of such debuggers is still work-in-progress.

Generating auxiliary lemmas is technically possible within some tactic languages, e.g. LTac (via the **abstract** command), but typically rather awkward: The lemmas generated by the tactic are ultimately inlined within the final proof, and thus not directly reusable within the proof of a different theorem. Also, tactic languages are "goal-oriented": While matching on certain internal terms is possible, one cannot directly use a tactic language for querying the AST of a given specification, which might be necessary for lemma generation or for deciding which tactics to apply.

Finally, all the existing tactic languages target the automation of general-purpose proofs. They do not allow for abstracting over the domain-specific aspects of a certain verification domain. This makes the usage of tactic languages rather inaccessible to domain experts: Firstly, the initial development of basic tactics for a verification domain is already a big hurdle, since it requires domain experts to learn a tactic language and to translate all the domain concepts they have in mind to the general-purpose constructs offered by the tactic language. Secondly, any domain expert or possibly end user who needs to inspect and/or refine the tactics because the basic tactics do not quite work in their particular case are faced with the inverse problem: They need to map the general-purpose constructs used in the tactics back to the domain concepts they know.

In summary, one could probably automate large parts of the proofs we are interested in by choosing one of the systems mentioned here and providing a collection of tactic scripts that encode high-level parts of progress and preservation proofs. However, firstly, there would be some aspects in the proofs we could not automate via tactic scripts, e.g. the generation of certain necessary auxiliary lemmas.

Secondly, the produced tactics would contain many prover-specific and internal commands and tactics. Thus, users of these tactics have to be very familiar with the prover used and with the chosen tactic language to be able to debug and extend a tactic script that fails for their specific soundness proof. Any domain-specific knowledge of a verification domain gets lost when using a general-purpose tactic language to automate a domain-specific verification task.

3.5. Summary

We learned about the main difficulty that arose when developing mechanized type soundness proofs for general-purpose languages, in the context of the PoplMark challenge: dealing with first-class binders in language specifications, which lead to the so-called "name-binding problem". There are solutions to the "name-binding problem", but automating them is not easy. Facing this observation, we restricted the set of DSLs that we consider in this thesis to DSLs without any first-class binders and argued that this subset of DSLs is interesting and relevant in practice.

We introduced an example DSL, namely a subset of typed SQL, and presented its specification and mechanized type soundness proof in two different existing verification systems (Isabelle/HOL and Dafny). Via these concrete proofs, we learned that obtaining them requires a certain level of expertise and effort on the user side. We also learned which abstract steps and auxiliary lemmas the type soundness proofs that we target require.

Finally, we analyzed how one could use existing verification systems to develop automatic strategies or tactics for generating type soundness proofs. We concluded that developing a suitable automation would require advanced skills in using tactic languages and/or in manipulating the internals of existing theorem provers. Also, the debugging and refinement of such tactics would not be a straightforward task. Most importantly, however, existing tools focus on general-purpose proofs and are insufficient for abstracting the domain-specific aspects of a verification task.

Chapter

4

VeriTaS: An Infrastructure for Domain-specific Verification

In Chapter 3, we saw which tools and methods currently exist for automating soundness proofs of type systems for DSLs. We laid out limitations of existing tools and which skills a researcher or developer needs to obtain to automate soundness proofs of type systems using these tools. In particular, the automation methods available within different tools are insufficient for abstracting the domain-specific aspects of a verification task.

In this Chapter, we introduce our own verification infrastructure, called VeriTaS[1], which we designed for simplifying proof automation in different verification domains. We start by deriving the requirements for such a verification infrastructure (Section 4.1). Next, we present *proof graphs* as a conceptual model on which we base the design of VeriTaS (Section 4.2).

We present a reference implementation of proof graphs as lightweight API in Scala, which provides a reusable verification infrastructure for domain-specific verification problems (Section 4.3).

Remark 4.1. This chapter contains content (especially sections 4.2.2 and 4.3) that the author of this thesis co-published in a paper at the international conference "Principles and Practice of Declarative Programming (PPDP)" in 2018, under the title "System Description: An Infrastructure for Combining Domain Knowledge with Automated Theorem Provers" [Gre+18b]. The author of this thesis co-published earlier versions of the vision for the VeriTaS system described in this chapter in 2015 [Gre+15] at the international conference "Onward!" and in 2016 [Gre16] at the "SPLASH Doctoral Symposium". ◊

[1] VeriTaS roughly abbreviates "**Veri**fication of **T**ype System **S**pecifications", since our target verification domain in this thesis are soundness proofs of type systems for DSLs.

4.1. Requirements for Domain-specific Verification

Having made the case that existing verification systems are not ideally equipped for supporting the needs of domain experts who want to fully automate a particular verification domain, we proceed with documenting the requirements for an infrastructure for domain-specific proof automation. The target user group of this verification infrastructure shall be domain experts as described in Section 1.3.2. We derive all requirements from our main motivation of developing an infrastructure for domain-specific verification.

4.1.1. Top-level Architecture

The overall architecture of the verification infrastructure shall accommodate the needs and skills of domain experts as given in Sections 1.3.2 to 1.3.3: We assume domain experts (notably within our target domain of soundness proofs of type systems for DSLs) to be fluent within some widespread general-purpose programming languages, but not necessarily with existing verification systems. Also, we identified that domain experts would like to be able to develop and use DSLs for verification within their domains as well as different individual input formats.

Requirement 4.1 (Library in general-purpose programming language with support for embedded DSLs)**.** The verification infrastructure should be designed and implemented as a library within a widespread general-purpose programming language. The chosen language should support the creation of embedded DSLs.

Designing the verification infrastructure as a library rather than as a standalone system allows for easily combining it with existing infrastructure and tools. Domain experts as well as experts in existing verification systems can easily extend a library with further high-level and low-level verification strategies, respectively. The usage of a widespread general-purpose programming language lowers the entry barrier for our target group: There typically exist good documentations for such a language as well as numerous well-developed IDEs that facilitate working with the language.

Using a programming language with good support for the creation of embedded DSLs allows domain experts to easily plugin their own formats for problem specifications as well as develop DSLs for implementing proof strategies within their domain. Having these custom languages embedded within the language of the verification infrastructure rather than as a standalone DSL facilitates the work flow for domain experts: They can work within a single language and environment when developing new formats and strategies or when extending existing ones. ◊

Requirement 4.2 (Parametric in input format)**.** The verification infrastructure, most notably the representation of proofs within the infrastructure, should be parametric in the format of problem specifications and proof obligations.

Leaving the input format as a parameter within the verification infrastructure enables domain experts to use existing as well as their own formats for specifying proof problems. Existing formats may come with features required in different domains, e.g., some formats may be executable, or are already integrated with

existing tools within the domain in question. A verification infrastructure parametric in the input format allows for easily reusing such a format along with any existing infrastructure. Domain experts may also develop their own formats as embedded DSLs (see previous requirement), along with such infrastructure, and plug them in. ◊

Requirement 4.3 (Decoupling of proof construction and sound step verification)**.** Domain experts should not have to deal with low-level technical concerns of verification systems when implementing their domain-specific proof strategies. Therefore, we require a strict separation of concerns between the parts of the verification infrastructure which a domain expert should normally have to touch and the parts which concern the "low-level" parts of verification. The latter should ideally only have to be touched by experts in different verification systems who would like to add support for their system or improve existing support.

We address this proposed separation of concerns between domain experts and verification experts by requiring that the construction of *high-level proof structures* shall be strictly decoupled from the verification of individual proof steps within a proof structure. This decoupling shall enable domain experts to focus on implementing domain-specific proof strategies for automatically generating the high-level parts of a proof, i.e., its overall structure together with the high-level strategies used. The verification of the individual steps outlined within a proof structure shall be independent from the generation of the structure itself.

We also require the infrastructure to allow domain experts to ignore the overall soundness of the proof structure when implementing automated strategies: They may generate proof steps that are incomplete or even incorrect. Our reasoning here is that the generation of an incomplete or incorrect proof structure, which contains nevertheless what one might call "ideas in the right direction", is beneficial for other domain experts and possibly end users when developing a proof in their verification domain: A slightly incorrect proof as a starting point is better than having nothing at all to start from.

Of course, the verification infrastructure should nevertheless enforce the overall soundness of the verification: Individual proof steps within a proof structure should only be marked as "verified" once a sound verification of that step took place, ideally via an external, established theorem prover. Therefore, the verification of individual proof steps is another important part of the verification infrastructure, decoupled from the construction of the proof structure itself.

The verification infrastructure shall compose these two parts (the generation of high-level proof structures and the sound verification of individual steps) in a way that safely allows for deducting the overall correctness of the generated high-level proof structure.

◊

4.1.2. Individual Features

The requirements we elaborated in Section 4.1.1 and our overall goal to design an infrastructure suitable for automating domain-specific verification tasks entail

several requirements for individual features within the verification infrastructure.

Requirement 4.4 (Interactive proof manipulation)**.** Above, we introduced the requirement that proof construction shall be decoupled from actual step verification, allowing for the generation of incomplete or even incorrect steps (Requirement 4.3). Hence, users of the verification infrastructure who want to prove a property using the automated strategies within some domain should be able to manipulate the generated proof in order to correct and refine generated proof steps.

The verification infrastructure should allow for the manual correction of all aspects of the generated proof: It should be possible to add new proof steps, modify proof strategies, modify existing steps and proof obligations/lemmas, and also to entirely delete generated steps. Users shall also be able to invoke the verification of manually changed parts of a proof and to obtain the result of this verification, i.e., proof manipulation should be *interactive*.

◊

Requirement 4.5 (Structured representation of proofs)**.** This requirement is directly connected to the previous requirement of enabling interactive manipulation of proof structures: To efficiently manipulate a proof structure, users have to be able to inspect the proof structure as quickly as possible.

Hence, a proof structure should be represented within the verification infrastructure in a structured, hierarchical fashion. It should be immediately clear from the representation of a proof structure how individual proof steps depend on each other, how one step can be proved from one or more other steps, and which steps were already successfully verified and which not. The representation should be in a human-readable format. Ideally, the verification infrastructure should provide an intuitive *visualization* of proof structures. ◊

Requirement 4.6 (Interface to different existing provers)**.** As emphasized above, we deliberately design a verification *infrastructure* rather than a standalone system. We deliberately aim at supporting the integration of different existing theorem provers, both automated theorem provers/SMT solvers and interactive theorem provers. The motivation for enabling support of a number of ATPs and ITPs is to obtain a high degree of automation for proofs, since different provers and solvers have different strengths. For example, SMT solvers are typically very strong at solving large, mostly ground problems, while ATPs typically handle quantified problems more elegantly [Bla12].

Hence, we require the verification infrastructure to contain clear interfaces which allow for connecting different existing provers for the verification of individual proof steps. In view of the assumed skills of our target group (Section 1.3.2), it shall be possible to connect existing provers in a way so that users of the verification infrastructure do not need to "leave" the verification infrastructure in order to work with the existing provers: It shall be possible to integrate automatic translations of input formats for the verification infrastructure into input formats for the existing provers, to call the provers from the interface, and to get a result from them back into the proof structure as represented within the verification infrastructure. ◊

Requirement 4.7 (Persistent proofs and verification states)**.** Our final requirement targets users who want to develop a proof using our verification infrastructure. Typically, proving individual steps with an ATP or SMT solver may take a lot of time (each prover may run several minutes on each step). Once users have reached a certain verification state, where some steps were successfully proven, they want to persist this state - either to continue later, or also to share the current state of the proof with other users who can continue then. They themselves or other users should not be forced to re-run provers on steps that were already successfully proven. Also, users should have a means to persuade themselves that the proofs found are indeed correct.

To this end, our verification infrastructure shall persist all verification states obtained (both for successfully proved steps and for steps where running external tools was inconclusive so far). Also, whenever possible, our verification infrastructure shall store any information from external provers about successfully discovered proofs. Ideally, one should be able to use this stored information for checking a proof step, be it manually or automatically. We call this additionally stored information *prover evidence*. Prover evidence may for example contain of a human-readable description of the proof found, or of a proof checkable by a trusted external prover or evidence checker. An *evidence checker* is a program that can reliably check given prover evidence for soundness. ◊

4.2. A Lightweight Representation of Proof Structures

We develop a conceptual, lightweight model for representing proof structures that satisfies our requirements from Section 4.1: We represent proof structures via proof graphs. We first give an informal introduction to proof graphs by example (Subsection 4.2.1). Next, we present a formal model of proof graphs and the related terminology and concepts that we use (Subsection 4.2.2). Finally, we argue how our model satisfies the requirements from Section 4.1 (Subsection 4.2.3).

4.2.1. Informal Introduction of Proof Graphs

We represent proof structures via proof graphs, a format inspired by previous work on proof planning, notably on the work of Richardson and Bundy [RB99]. However, in our model, we shift the focus away from tactic-based proof plans toward proof structures represented primarily by the intermediate subgoals that are generated during a proof. We discuss the differences to Richardson and Bundy's work further in Section 9. Furthermore, we develop concepts for constructing and verifying proof graphs.

Our proof graphs are directed acyclic graphs (DAGs), whose nodes consist of intermediate proof obligations along with at least all the specifications of datatypes, functions, and predicates directly or indirectly used within the proof obligation. The nodes are parametric in the format of the proof obligations and of the specification. Root nodes represent top-level theorems, leaf nodes represent proof obligations in the proof which trivially follow from the specification. Nodes in proof graphs

are connected by directed proof edges that indicate the dependencies between the intermediate proof obligations. Proof edges may be labeled with additional, proof-relevant information from parent steps (e.g. induction hypotheses). Nodes in proof graphs have tactics associated which indicate *how* a parent proof obligation follows from its child proof obligations within the proof graph, or, in the case of leaves, how the proof obligation can be proven directly from the specification. From this representation, individual proof steps can be derived, which can then be individually and independently of other proof steps translated to individual proof problems that can be sent to external provers for verification. Developers may implement proof strategies for automatically constructing parts of or entire proof graphs.

We informally introduce the concepts just mentioned by example: We show how to construct a proof graph that represents the rough proof structure for proving progress of typed arithmetic expressions (see Section 2.2 and Section 2.3, where we presented typed arithmetic expressions and a type soundness proof for the associated type system, as introduced by Pierce [Pie02]).

We show a proof graph for our example proof in Figure 4.1. Boxed nodes represent proof obligations, diamond-shaped nodes proof steps containing tactics. The root node, labeled "Progress", contains a specification of typed arithmetic expressions (syntax, semantics, and type systems), together with the top-level progress theorem to prove (the proof obligation). We apply the tactic "StructuralInduction", which yields 7 sub-proof obligations: One induction case for each possible arithmetic expression, each again containing the full specification of typed arithmetic expressions. The associated proof edges contain the induction hypotheses where applicable. For example, the "ProgressIfelse" edge contains three induction hypotheses for the guard-expression, the then-expression, and the else-expression, the "ProgressSucc" edge contains one induction hypothesis for the expression within a *Succ* construct, and the "ProgressFalse" edge contains no induction hypothesis, since this case is a base case. We can first apply a tactic named "Solve" to all our induction cases. This tactic simply designates that we expect an external prover to be able to verify this case from the given specification only, translating each invididual proof step independently to an proof problem for an external prover.

In Figure 4.1, we show how an intermediate verification state could look like: We may send the proof steps of the three base cases ("ProgressTrue", "ProgressFalse", and "ProgressZero") to external verifiers and find that they can easily be verified. Hence, the corresponding "Solve" steps will be marked with green boxes within a proof graph, visualizing the step result. Since these steps did not have any children themselves, nothing remains to be done for verifying the base cases, hence the corresponding boxed nodes are marked green as well. Next, we try to apply an external verifier to the "Solve" proof step of an induction case, "ProgressSucc". The verifier we applied does not return a conclusive result, hence the proof step as well as the parent proof obligation is marked in red. Nodes marked in grey indicate that we did not yet attempt to apply any verifier to a step.

For the "ProgressIfelse" case, Figure 4.1 gives an example of another tactic application: Here, we apply a tactic named "CaseDistinction" to the "ProgressIfelse" proof obligation, distinguishing three cases, which become sub-proof obligations of

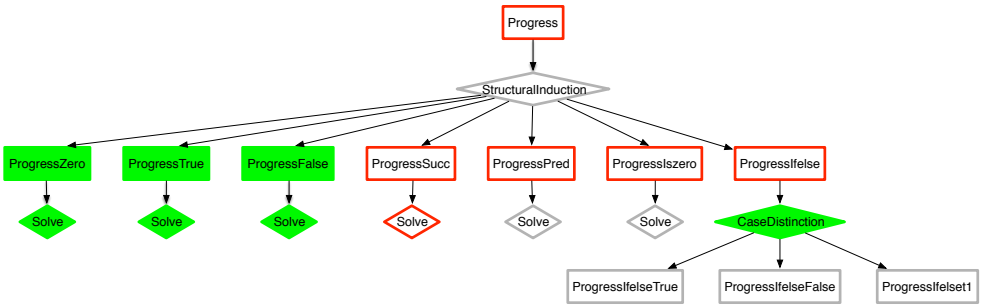


Figure 4.1.: A first proof graph for the progress proof for the type system for typed arithmetic expression, with an intermediate verification state

the corresponding proof step. Next, we translate the proof step marked "CaseDistinction" to a proof problem: This problem will now contain the specification of typed arithmetic expressions as well as the three refined obligation statements from the three sub-cases, as axioms. The intuitive semantics of the proof problem is thus: "Provided the cases all hold individually, can we prove the parent obligation?" Here, our verifier can do that, hence the "CaseDistinction" proof step is marked green. However, in the intermediate state in Figure 4.1, we did not yet apply any verifier to the three sub-proof obligations (hence marked in grey), so we do not yet know if our parent proof obligation "ProgressIfelse" really holds, which is hence marked in red. If we obtain complete proofs from external verifiers for the three sub-cases as well, we can mark the "ProgressIfelse" obligation in green. Note that induction hypotheses are carried along the proof edges, e.g. edge "ProgressIfelsecaseFalsecase" will contain all three induction hypotheses from the "ProgressIfelse" case, which will be included when translating a (currently missing) proof step for this case to a proof problem.

We will revisit the proof graphs for this example in Chapter 6.

4.2.2. Formal Model of Proof Graphs and Related Concepts

We formally define the concepts introduced by example in the previous section. We first define the formal structure of proof graphs, then formally introduce concepts and terminology for constructing proof graphs, and finally for verifying proof graphs.

Definition of Proof Graphs

The nodes of our proof graphs are proof obligations.

Definition 4.1 (Proof obligation)**.** A *proof obligation* consists of a *problem specification*, i.e. a set of definitions and axioms together with a conjecture to prove. A proof obligation $p_{\mathcal{D},\mathcal{G}}$ is parametric in a format $\mathcal{D}$ for definitions and in a format $\mathcal{G}$ for proof goals/conjectures. We denote a set of proof obligations parametric in $\mathcal{D}$ and $\mathcal{G}$ with $\mathcal{P}_{\mathcal{D},\mathcal{G}}$. ◊

To avoid notational clutter, we omit the subscripts $\mathcal{D}$ and $\mathcal{G}$ from now on and just write p (occasionally with numerical subscripts) for proof obligations and $\mathcal{P}$ for a set of proof obligations parametric in $\mathcal{D}$ and $\mathcal{G}$.

Proof obligations are connected to each other via directed proof edges.

Definition 4.2 (Proof edge)**.** A *proof edge* is a triple $(p_1,\ l,\ p_2)$ from a proof obligation p_1 to another proof obligation p_2. A proof edge additionally carries an *edge label* l from a set of possible edge labels $\mathcal{L}$. We denote a set of proof edges labeled with elements of $\mathcal{L}$ with $\mathcal{E}_\mathcal{L}$. ◊

Since proof edges are directed, a proof obligation can have children, or *sub-obligations*. The semantics of sub-obligations is that proving the parent obligation may require proving some or all of the sub-obligations, or, differently put, that the proof of the parent obligation may depend on proving the sub-obligations.

Edge labels may be used for different purposes, for example to propagate proof-relevant information from parent obligations to sub-obligations. Such proof-relevant information could include fixed variables or induction hypotheses from the parent obligation which also apply for proving the sub-obligations.

We define the structure of proof graphs using proof obligations and proof edges.

Definition 4.3 (Proof graph)**.** A *proof graph* is a directed acyclic graph (DAG) $(\mathcal{P},\ \mathcal{E}_\mathcal{L})$, i.e. with nodes from a set of proof obligations $\mathcal{P}$ and edges from a set of labeled proof edges $\mathcal{E}_\mathcal{L}$. We denote the set of all proof graphs, i.e. the type of proof graphs, with PG. ◊

Note that proof graphs have to be acyclic in order to represent correct proofs. We use graphs instead of trees in order to allow for proof obligations to be reused within a proof. For example, a sub-obligation may consist of an auxiliary lemma which is required in different parts of a proof. Rather than duplicating the proof obligation within a tree (including the sub-tree which models the proof of the auxiliary lemma), we allow for sub-obligations to have more than one parent obligation.

Proof graphs can have one or more *root obligations*, i.e. proof obligations without a parent obligation. Root obligations model the theorems one wants to prove. *Leaf obligations* are proof obligations without any sub-obligation. A leaf obligation models a theorem or lemma which trivially follows from the definitions within the problem specification of the leaf obligation.

Constructing Proof Graphs

Next, we formally introduce concepts and terminology for the manual and/or automated construction of proof graphs. We may apply a tactic to a proof obligation in order to add a single proof step to that proof obligation.

Definition 4.4 (Tactic)**.** A *tactic* t is a function of type $(\mathcal{P} \times \overline{\mathcal{E}_\mathcal{L}}) \to \overline{(\mathcal{P} \times \mathcal{E}_\mathcal{L})}$, where $\mathcal{P}$ denotes the set of all proof obligations (for a given format for definitions $\mathcal{D}$ and a given format for goals $\mathcal{G}$) and $\mathcal{E}_\mathcal{L}$ denotes the set of all labeled edges. We denote the set of all tactics with T. ◊

A tactic takes a proof obligation and a list of proof edges, namely the incoming edges of the given proof obligation, as an argument and returns a list of pairs of a proof obligation and a proof edge. Applying a tactic to a proof obligation deterministically generates a list of sub-obligations with corresponding proof edges to each of the sub-obligations. The tactic requires the list of incoming edges to the given proof obligation in order to generate new proof edges and correctly propagate information if necessary. A tactic may also return an empty list. For example, the simplest tactic, which we name *Solve*, simply declares that a proof obligation can be proven directly via its problem specification. Hence, applying *Solve* does not generate any further proof obligations. Additionally, if the tactic is not applicable to the given parent obligation, the function t also returns the empty list.

Definition 4.5 (Proof step)**.** A *proof step* s is a triple $(p,\ t,\ \overline{(p, e_{\mathcal{L}})})$. We denote a set of proof steps with $\mathcal{S}$. ◊

The leftmost element of the triple represents the parent obligation p, the second element the tactic t applied to p, and the last element the list of sub-obligations with corresponding labeled proof edges $e_{\mathcal{L}}$ generated by t for p and its incoming edges. For each parent obligation p in a proof graph, there may be at most one proof step.

Applying a tactic t within a proof graph pg_1 to a parent obligation p generates a proof step for p and returns a proof graph pg_2 augmented by the proof edges and sub-obligations generated by applying t to p and its incoming edges, together with the generated proof step s; formally:

Definition 4.6 (Tactic application)**.** The function *applyTac* of type $(PG\ \times\ \mathcal{P} \times T) \rightarrow (PG\ \times\ \mathcal{S})$ is defined so that $applyTac(pg_1, p, t) = (pg_2, s)$, where

- $pg_1 = (\mathcal{P}_1, \mathcal{E}_{\mathcal{L},1})$
- $pg_2 = (\mathcal{P}_2, \mathcal{E}_{\mathcal{L},2})$
- $t(p, e_{\mathcal{L}}) = ((p_1, e_{\mathcal{L},1}), ..., (p_n, e_{\mathcal{L},n}))$
- $s = (p, t, ((p_1, e_{\mathcal{L},1}), ..., (p_n, e_{\mathcal{L},n}))\ )$
- $\mathcal{P}_2 = \mathcal{P}_1 \cup \{p_1, ..., p_n\}$
- $\mathcal{E}_{\mathcal{L},2} = \mathcal{E}_{\mathcal{L},1} \cup \{e_{\mathcal{L},1}, ..., e_{\mathcal{L},n}\}$,

if $p \in \mathcal{P}_1$. Otherwise, $pg_1 = pg_2$, i.e. the given proof graph remains unchanged.[2]

◊

Tactics can only ever be used to construct a single proof step for a given proof obligation. Proof strategies heuristically generate larger parts of or entire proof graphs:

Definition 4.7 (Proof strategy)**.** A *proof strategy Str* is a function $PG \rightarrow PG$, i.e. a function from a proof graph to a proof graph. ◊

[2] An implementation of *applyTac* may return appropriate messages.

Initially, a proof strategy may be applied to a proof graph that only consists of one or more root obligations without any sub-obligations. Proof strategies may apply other proof strategies and/or tactics in order to construct proof graphs. But most importantly, a proof strategy may operate *globally* on a proof graph and the problem specifications within its proof obligations, while tactics only ever operate *locally* on a given proof obligation. Thus, proof strategies may for example heuristically generate sub-obligations that contain auxiliary lemmas and insert them at certain points of the given proof graph. Users may manually apply tactics as well as single proof strategies to refine a proof graph.

Verifying Proof Graphs

The proof graphs constructed by proof strategies and/or via tactic application represent suggested proof structures and may contain unprovable proof obligations and incorrect proof steps. To actually verify a proof graph, each of its proof obligations has to be verified with an external verifier. We formally define what "verifying a proof obligation" means.

To verify a proof obligation, one has to attach a proof step to the proof obligation by applying a tactic. Next, an external verifier has to verify the attached proof step. A proof step is verified by first encoding the proof step as a proof problem in a verifier format $\mathcal{V}$ and then applying a verifier onto the proof problem in order to obtain a step result.

Definition 4.8 (Encoding proof problems)**.** A function *enc* of type $\mathcal{S} \to \mathcal{V}$ encodes a proof step into a *proof problem* in a verification format $\mathcal{V}$. ◊

Definition 4.9 (Step result)**.** A *step result* is a triple $(s, stat, ev)$, where s is a proof step, *stat* is a proof status label that has one of the textual values $\{Proved,\ Disproved,\ Inconclusive\}$, and *ev* is any kind of evidence for the verifier's result. We denote a set of step results with $\mathcal{R}$. ◊

The evidence for a proof result may, for instance, be a proof (for *Proved* results), a counterexample or contradiction (for *Disproved* results), or it can also be empty (e.g. for *Inconclusive* results). For example, some TPTP provers [Sut17] produce proofs in the TSTP format [Sut10], which we could use as evidence for *Proved* results.

Definition 4.10 (Verifier)**.** A *verifier* is a function *ver* of type $\mathcal{V} \to \mathcal{R}$ that produces a step result $(s, stat, ev)$ when given a proof problem in a verification format $\mathcal{V}$. ◊

Definition 4.11 (Verifying a proof step)**.** A proof step s is verified if, for a verifier *ver* and an encoding function *enc*, $ver(enc(s)) = (s, Proved, ev)$. ◊

The verification of a proof step is only the first part of verifying a proof obligation. To fully verify a proof obligation, we recursively have to verify all its sub-obligations:

Definition 4.12 (Verifying a proof obligation)**.** A proof obligation p in a proof graph $(\mathcal{P}, \mathcal{E}_\mathcal{L})$ (i.e. $p \in \mathcal{P}$) is verified if both of the conditions below hold:

1. The unique proof step $s = (p,\ t,\ ((p_1, e_1), ..., (p_n, e_n))\ )$ is verified.

2. All the sub-obligations of p are verified, i.e. $p_1, \ ..., \ p_n$.

$\Diamond$

A proof graph only represents a fully correct proof if all its proof obligations are verified. The proof graph itself only serves as a means to structure and assemble the individual proof problems within a proof. The individual proof problems represented by the proof steps may be verified by different ATPs and SMT solvers.

4.2.3. How Proof Graphs Satisfy Our Requirements

We argue why we chose to model proof structures using proof graphs and how our model satisfies our requirements for a verification infrastructure for automated domain-specific verification (Section 4.1).

Our conceptual model of proof graphs is lightweight and general enough that it can easily be implemented within any suitable general-purpose programming language. In Subsection 4.3.1, we motivate our language choice (Scala) with regard to Requirement 4.1.

The nodes of our proof graphs, i.e. the proof obligations, are parametric in a format $\mathcal{D}$ for definitions (for problem specifications) and in a format $\mathcal{G}$ for the goals of a proof (as modeled in Definition 4.1). Thereby, we satisfy Requirement 4.2: By instantiating $\mathcal{D}$ and $\mathcal{G}$ in proof graphs, developers may use our verification infrastructure with a number of different custom formats. By providing corresponding encodings of proof problems for these custom formats (Definition 4.8), developers can integrate their formats fully with the verification infrastructure and existing verifiers.

Next, we defined proof graphs so that proof construction and step verification are decoupled from each other, as asked for in Requirement 4.3: Proof construction is done by implementing proof strategies (Definition 4.7), which may construct proof graphs without step results. Step verification is done by calling a verifier (see Definition 4.10) on an encoded proof step, as formalized in Definition 4.11. The proof graphs themselves strictly only model proof structures where every proof obligation has to be verified in order for the proof graph to represent a correct proof. The verification of all proof obligation entails verifying the corresponding proof steps with separate verifiers (Definition 4.12).

Algorithms for manipulating directed acyclic graphs (inserting nodes, replacing nodes, deleting nodes etc.) are widely known and easy to implement. Hence, we can easily satisfy our Requirement 4.4 (interactive proof manipulation) by implementing such operations on proof graphs and allowing users to invoke the verification of individual proof steps via an API. Additionally, directed acyclic graphs are by their very nature structured and easily visualizable. We may visualize the verification states of individual proof steps within proof graphs for example via colors, enabling users to easily understand proof structures and verification states, as described in Requirement 4.5.

By implementing calls to external ATPs, ITPs, and SMT solvers as verifiers (Definition 4.10) within our verification infrastructure, we may connect numerous existing verification systems to our verification infrastructure, as described in Re-

quirement 4.6. We may implement different encoding strategies for encoding proof problems for external verifiers (Definition 4.8).

Finally, proof graphs may easily be serialized for persisting proofs. Within step results, we save prover evidence returned from verifiers upon successful verification of proof steps. We may easily serialize this evidence along with a proof graph, thereby also persisting verification states. Thus, our model also satisfies Requirement 4.7.

4.3. A Proof Graph API in Scala

We implement our model of proof graphs (see Section 4.2) as an API in Scala. Our API is versatile and can be flexibly instantiated: Domain experts may provide their own instantiations for concrete data structures to represent proof graphs, their own input formats, their own tactic implementations, their own verifiers and encodings of proof problems, and of course their own proof strategies for automated proof construction.

In this section, we describe our API for proof graphs and its implementation within VeriTaS. We also present a reference implementation of proof graphs using a transactional schema-less database as the underlying data structure (see Subsection 4.3.3). Our implementation of VeriTaS is publicly available at `https://github.com/stg-tud/type-pragmatics/tree/master/Veritas`.

4.3.1. Why Scala?

We introduced the main aspects and features of Scala in Section 2.4. Here, we restate some of Scala's features that are advantageous for our purposes and link them with Requirement 4.1.

Scala is fully compatible with Java. Hence, implementations in Scala can run on any platform for which there is a JVM. Furthermore, one can use all existing Java libraries within Scala. Thus, a verification infrastructure implemented in Scala can easily be used with a lot of other existing libraries and tools. For instance, there are special-purpose ATPs implemented within Scala or Java, such as princess [Rüm18; Rüm08a], which targets problems in Presburger arithmetic. There is also a Scala library for interfacing with Isabelle[3] developed by Lars Hupel. There are numerous ongoing research projects in the area of programming languages that were or are currently being implemented in Scala (e.g. [Pap+11; Don18; Hau+16], to name only a very small selection). Hence, it is a reasonable assumption that knowledge of Scala is widespread on the side of domain experts (notably, in the area of programming languages) as well as on the side of experts for existing verification systems. Scala can be used within several widespread IDEs, e.g. IntelliJ IDEA[4] and Eclipse[5], which further facilitates working with the language.

Furthermore, Scala is very well equipped for the creation of embedded DSLs, which was part of our requirements for the underlying programming language of

[3] `https://github.com/larsrh/libisabelle`
[4] `https://www.jetbrains.com/idea/`
[5] `http://www.eclipse.org/`

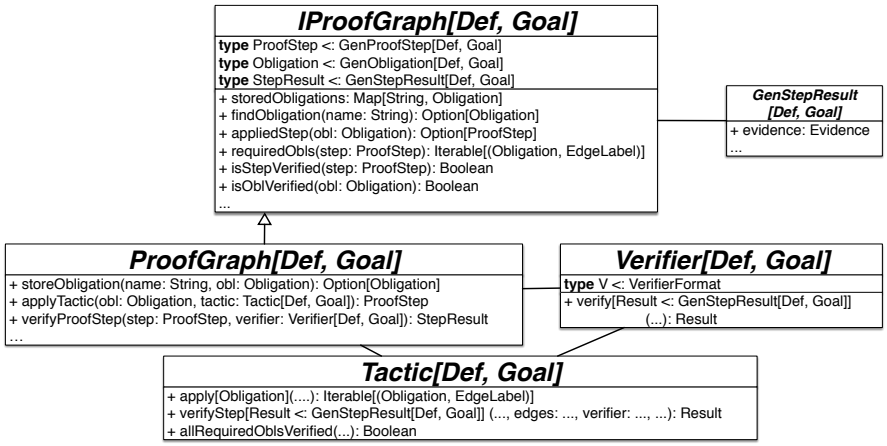


Figure 4.2.: Overview of core API for proof graphs within VeriTaS

our verification infrastructure (see Requirement 4.1): Scala allows for using almost arbitrary symbols as names of methods and classes and provides *implicit* methods and classes. These two features together can be used for creating embedded DSLs with the look and feel required for particular domains. Some examples of how to design small embedded DSLs within Scala may be found in the book "Programming in Scala" [OSV11]. We will also provide an example in Subsection 5.1.1.

4.3.2. Modeling Proof Graphs Via Scala Traits

Figure 4.2 summarizes the main Scala traits of our core API, listing the most important API methods for illustration.

All components of the core API are parametric over a format for definitions (type parameter `Def`) and over a format for proof obligations (type parameter `Goal`), as formalized in Section 4.2. The main components of VeriTaS are the two traits `IProofGraph` and `ProofGraph`.

The trait `IProofGraph` (= *immutable* proof graph) defines the components of a proof graph and groups all read-only methods on proof graphs. The trait defines type members for proof steps, proof obligations, and step results. Each of these type members extends a corresponding generic trait (prefix `Gen`), which defines a few basic, minimal fields for proof steps, obligations, and step results. Concrete implementations of trait `IProofGraph` may provide custom data structures to instantiate the type members of `IProofGraph`. We will describe an example instantiation in Subsection 4.3.3.

One of our design conventions is that all main obligations of a proof, and especially all root obligations, are stored with a string name. One can access a map of all stored obligations and their names via method `storedObligations` and access a single obligation via its string name using method `findObligation`. Given an obligation, one can access its proof step via method `appliedStep`, if the obligation

has a proof step attached. Given a proof step, one can access the attached edge labels and sub-obligations via method `requiredObls`. Methods `isStepVerified` and `isOblVerified` allow for querying whether a proof step resp. a proof obligation in a proof graph is verified or not. Trait `IProofGraph` also contains further API methods which we omitted in Figure 4.2, for example methods which allow for accessing the parents of an obligation.

Trait `ProofGraph` extends `IProofGraph` with methods for modifying a proof graph. One may add a new proof obligation to a proof graph via method `storeObligation`, assigning a custom name to the obligation object. Most importantly, one can grow a proof graph by applying a `Tactic` to one of the graph's `Obligation`s (method `applyTactic`). This will add a proof step and sub-obligations to the graph, but it will not yet verify that step. This way, we implement requirement 4.3 of decoupling proof construction and step verification: Proof strategies can construct the entire structure of a proof before any potentially unsuccessful verification is started. Thus, one may program proof strategies that do not get immediately stuck if a single verification step fails along the way. The verification of a proof obligation and potential correction of a generated proof graph is prover-specific and may require user intervention.

One can verify a proof step via method `verifyProofStep`. This method takes the proof step to verify and a verifier as arguments. A verifier has to implement trait `Verifier`, which has a type parameter `VerifierFormat` that can be instantiated as needed for a specific verifier. Trait `Verifier` consists of a method `verify` that has to produce a step result, i.e. an instance of `GenStepResult`. `GenStepResult` contains a field `evidence` that the verifier has to deliver. The format for evidence is prover-specific and so are evidence checkers. We store the evidence for each proof step in the graph, such that it becomes possible to check proofs developed by others. A concrete implementation of `Verifier` may for example consist of a custom verifier implemented within Scala, or call existing external ATPs and SMT solvers, translating their results into an instance of `GenStepResult`. We will see examples of concrete `Verifier` instantiations in Subsection 5.3.

Method `verifyProofStep` in trait `ProofGraph` calls method `verifyStep` within the `Tactic` instance stored in the given proof step. Method `verifyProofStep` passes the given verifier on to `verifyStep`, along with the parent edges that may contain information which could be needed for verifying the proof step, such as induction hypotheses or fixed variables. Method `verifyStep` first translates a proof problem in format `Def` and `Goal` to the `VerifierFormat` of the given verifier, then finally calls method `verify` of the given verifier and passes the translated problem.

ProofGraph extensions

To augment the proof graph core API with additional functionality, we created further utility traits and classes: First, we created a trait `ProofGraphVisualizer` for visualizing a given proof graph. This trait contains methods for encoding obligations, proof steps, and edges into a format that can be visualized. As an example, we implemented trait `ProofGraphVisualizer` via class `GraphVizVisualizer`, which

translates a given proof graph into the dot format so that the graph can be visualized using GraphViz.[6]

Second, we created a trait `ProofGraphTraversals`, which extends `ProofGraph` with additional traversal methods for traversing obligations and proof steps, as well as for applying fold and map operations on obligations etc.

Third, we created a class `ProofGraphUI`, which allows for more convenient access of inner sub-obligations of a proof graph: One can pass a function when instantiating `ProofGraphUI` which calculates a string for a given obligation, and then one can access inner obligations in a given proof graph via the calculated name.

4.3.3. A Reference Implementation of Proof Graphs

We implemented trait `ProofGraph` using the Java library for databases Xodus[7] for persisting proof graphs. Xodus is a library for transactional schema-less databases developed by JetBrains: The class `ProofGraphXodus` implements obligations, proof steps, and step results as entities within a Xodus database with links among each other. The fields of obligations, proof steps, and step results either become properties of the corresponding entities, or separate entities. For example, the goal of an obligation becomes a property of an obligation entity. However, we create separate entities for specifications, which in turn save the actual specification within a property. An obligation entity then links to a specification entity within the database. This has the advantage that we can link to one specification entity from many different obligation entities and hence improve on the overall size of the database: It is likely that for many problems, specifications will be large and that many obligations within a single proof graph will have the same specification. Hence, it makes sense to let several or even all obligations share a specification internally.

We implemented the methods inherited from trait `IProofGraph` via read-only transactions and the additional methods from trait `ProofGraph` via regular transactions. By using transactions, we ensure that edits of a proof graph do not lead to an inconsistent state, even when processing a proof graph concurrently.

Our reference implementation of proof graphs is still configurable in the concrete format for obligations, proof steps, step results etc. Hence, our reference implementation may be used with many different verification domains.

4.4. Summary

We first derived concrete requirements for a verification infrastructure suited for the automation of different verification domains by domain experts. The key requirements for appropriately meeting the skills of domain experts that we outlined in Subsection 1.3.2 are, on the abstract side, the decoupling of the generation of proof structures from the verification of individual proof steps, and on the technical side, to provide a generic implementation of the verification infrastructure as an API in a known general-purpose programming language.

[6] `http://www.graphviz.org/`
[7] `http://jetbrains.github.io/xodus/`

Next, we presented a formal model of a verification infrastructure that satisfies our requirements, via proof graphs. We defined proof graphs to be generic in a format for problem descriptions. Notably, in our definition of proof graphs, we separated of obligations (nodes of a proof graph) from verifiable proof steps (edges of a proof graph), thereby decoupling the generation of proof structures from the verification of individual proof steps as required. Finally, we presented a generic as well as a concrete implementation of proof graphs as an API in Scala, meeting our technical requirements.

Thus, we now have obtained a generic verification infrastructure suitable for automating our target verification domain of type soundness proofs for DSLs, but also various other verification domains.

Chapter

5

An Instantiation of VeriTaS for Type Soundness Proofs of DSLs

We instantiate our generic VeriTaS verification infrastructure from Chapter 4 to create type soundness proofs of DSLs interactively. The different components of this instantiation serve as the basis for the automated proof strategies which we will present in Chapter 6. First, we create a suitable input format for specifications of type systems (Section 5.1). Next, we implement basic tactics that enable us to construct proof steps requires in type soundness proofs (Section 5.2). We implement these tactics so that they may be reused in other verification domains and with other input formats. Finally, we connect different existing ATPs and SMT solvers to VeriTaS that enable us to verify proof steps (Section 5.3). A crucial part of connecting external verifiers is to encode a given input format into the formats used by the ATPs and SMT solvers. Thus, we present suitable encodings from our core input format into TPTP (for ATPs) and SMTLIB (for SMT solvers) in Sections 5.4 and 5.5).

Remark 5.1. The author of this dissertation co-published content from the first part of this chapter (in particular, from Sections 5.1, 5.2, and 5.3) in a paper at the international conference "Principles and Practice of Declarative Programming (PPDP)" in 2018 [Gre+18b]. Furthermore, the author of this thesis co-published content from Section 5.4 in a paper within the journal "Science of Computer Programming" in 2018, under the title "Exploration of language specifications by compilation to first-order logic" [Gre+18a]. An earlier version of this journal paper appeared in the conference "Principles and Practice of Declarative Programming (PPDP)" in 2016, under the same title [Gre+16]. In this thesis, the content from the journal paper was updated and adapted to the terminology and examples used within the thesis. The material on ScalaSPL (Section 5.1.2) builds on the master's thesis of Pacak [Pac18], which the author of this dissertation supervised. ◊

5.1. Input format

We develop an input format that allows for defining simple type system specifications and properties of type systems. Our input format shall be able to express a language's syntax via algebraic data types (ADTs), a language's reduction semantics via simple recursive functions, and a type system via inference rules, in a format similar to the format introduced in Section 2.2) Furthermore, the format should provide a means to express properties such as progress and preservation of type systems, together with auxiliary lemmas needed for proving these properties. These minimal ingredients are sufficient for domain experts in type soundness proofs to express a language's syntax as well as its static and dynamic semantics: As explained in Subsection 1.3.2, we assume our target domain experts have knowledge of functional programming languages, which typically provide ADTs and recursive functions.

We first present a core specification language (called SPL) which may be used in different verification domains in Subsection 5.1.1 and an input format that targets type soundness proofs specifically and is based on SPL and Scala in Subsection 5.1.2.

5.1.1. SPL: A Core Specification Language

We design a core specification language for VeriTaS, which we call SPL (short for "specification language"). Our language features constructs for representing ADTs, simple recursive functions, and a notation for inference rules. To keep SPL a core language without too many extra constructs, we overload the inference-rule notation to be used both for representing typing rules and for representing axioms and properties on type system specifications. Since SPL is a core language, it may be used as an intermediate format by other input formats, as we demonstrate in Subsection 5.1.2. This allows for reusing existing infrastructure for SPL such as the encodings of SPL to ATP input formats (see Section 5.4).

We introduce the constructs of SPL via our running example of typed arithmetic expressions. Internally, all language constructs of SPL are implemented via case classes in Scala, which all extend a common trait `VeritasConstruct`. The constructs which one can use to express properties extend a common trait `VeritasFormula`, which is a sub-trait of `VeritasConstruct`. Hence, to create a proof graph using SPL as input format, one simply has to create an instance of `ProofGraph[VeritasConstruct, VeritasFormula]`. To facilitate reading the example specifications, we present all examples within a custom, human-readable format rather than with the corresponding case class instances. We mention the names of the corresponding top-level case classes along with the examples. Our implementation offers a pretty-printer for terms of type `VeritasConstruct` that prints them in the same human-readable format we use in this thesis.

Algebraic Data Types (ADTs)

SPL supports closed algebraic data types and open data types for the definition of a language's syntax. Open data types may be used for declaring underspecified data types, whose structure is irrelevant for a type soundness proof. For instance, we

may introduce an open datatype Val to represent values within cells of tables like this:

```
1  open data Val
```

Listing 5.1: Open data types in SPL

Within the code base of VeriTaS, we represent open data types with the case class `DataType`, which takes as arguments a Boolean to mark whether the data type is open or not (true for an open data type, false for a closed one), the name of the data type, and a list of data type constructors (case class `DataTypeConstructor`), which would be empty for Val. Open datatypes in SPL are countably infinite.

Our running example of typed arithmetic expressions does not require under-specified data types, but only classical closed ADTs, which have a fixed number of constructors. For example, we would specify the syntax of typed arithmetic expressions ("terms") like in Listing 5.2.

```
1 data Term =
2   true | false | ifelse(Term, Term, Term) |
3   zero | succ(Term) | pred(Term) | iszero(Term)
4
5 consts t1: Term; t2: Term
```

Listing 5.2: Syntax of typed arithmetic expressions in SPL

Data type Term has seven constructors, separated by |: true, false, and zero, which have no constructor arguments, and succ, pred, iszero, and ifelse, which have recursive constructor arguments. Via the **consts** construct (line 5), one can introduce names for instances of closed or open data types, e.g. for describing concrete terms (implemented by case class `Consts`).

Recursive Functions

For the definition of a language's dynamic semantics, SPL supports partial and total first-order function definitions. For example, we can define the dynamic semantics of typed arithmetic expressions as a deterministic small-step reduction function reduce as in Listing 5.3.

```
1  data OptTerm =
2      noTerm | someTerm(Term)
3
4  function isSomeTerm: OptTerm → Bool
5      isSomeTerm(noTerm) = false
6      isSomeTerm(someTerm(t)) = true
7
8  partial function getTerm: OptTerm → Term
9      getTerm(someTerm(t)) = t
10
11 //reduction semantics for simple Boolean and arithmetic terms
```

```
12 function reduce: Term → OptTerm
13     reduce(ifelse(true, t2, t3)) = someTerm(t2)
14     reduce(ifelse(false, t2, t3)) = someTerm(t3)
15     reduce(ifelse(t1, t2, t3)) =
16       let ot1 = reduce(t1)) in
17           if (isSomeTerm(ot1))
18             then someTerm(ifelse(getTerm(ot1), t2, t3))
19             else noTerm
20     reduce(succ(t1)) =
21       let ot2 = reduce(t1)) in
22           if (isSomeTerm(ot2))
23           then someTerm(succ(getTerm(ot2)))
24           else noTerm
25     reduce(pred(zero)) = someTerm(zero)
26     reduce(pred(succ(nv))) =
27       if (isNV(nv))
28       then someTerm(nv)
29       else let ot2 = reduce(succ(nv)) in
30           if (isSomeTerm(ot2))
31             then someTerm(pred(getTerm(ot2)))
32             else noTerm
33     reduce(pred(t1)) =
34       let ot2 = reduce(t1)) in
35           if (isSomeTerm(ot2))
36             then someTerm(pred(getTerm(ot2)))
37             else noTerm
38     reduce(iszero(zero)) = someTerm(true)
39     reduce(iszero(succ(nv))) =
40       if (isNV(nv))
41       then someTerm(false)
42       else let ot2 = reduce(succ(nv)) in
43           if (isSomeTerm(ot2))
44           then someTerm(iszero(getTerm(ot2)))
45           else noTerm
46     reduce(iszero(t1)) =
47       let ot2 = reduce(ot2)) in
48           if (isSomeTerm(ot2))
49           then someTerm(iszero(getTerm(ot2)))
50           else noTerm
51     reduce(t) = noTerm
```

Listing 5.3: Semantics of typed arithmetic expressions in SPL

In line 1 and 2, we first define an option type for Term, which we use as return type for reduce to mark that reductions of terms may fail. Functions isSomeTerm (starting line 4) and reduce (starting line 12) are total functions (case class `Functions`), that is, they yield a result for any well-typed input. In contrast, function getTerm (lines 8 and 9) has been declared partial (case class `PartialFunctions`) because it only yields a result for a subset of its inputs (terms of the form someTerm(t)).

Function definitions (both for total and partial functions) start with the function's type signature (case class `FunctionSig`), which defines the types of the function's

arguments and its return type. The function definition consists of a list of function equations (case class `FunctionEq`): The left-hand side of each function equation contains a pattern for the function's arguments (case class `FunctionPattern`), the right-hand side of a function expression contains the function expression (case class `FunctionExp`) which defines the function's behavior for this pattern. Function patterns may contain *variables* (case class `FunctionPatVar`), which match any term and may be used within the defining function expression. The order of the function equations matters for the semantics of SPL: The function's arguments are matched against the patterns of the function equations from top to bottom, the first pattern that matches counts. Hence, the default case reduce(t) in line 45 matches only patterns which none of the previous patterns cover, e.g. t = zero. Note that defining a default case is necessary for the total function reduce, since its defining patterns do not cover all possible patterns for Terms. We do not need a default case for function isSomeTerm: There, the two function equations cover all possible patterns for type OptTerm, so isSomeTerm is total.

Top-level function expressions used within reduce are function applications (case class `FunctionExpApp`), let expressions (case class `FunctionExpLet`), and if expressions (case class `FunctionExpIf`). Beyond these, SPL also supports standard Boolean expressions (equations, negations, conjunctions, disjunctions, etc.). Note that for simplicity, SPL does not differentiate between constructor applications and function applications - both are covered by case class `FunctionExpApp`.

Inference Rules and Properties

For the definition of typing rules as well as properties, SPL supports the inductive definition of relations via inference rules (case class `TypingRule`). Typing rules accept function expressions as arguments as well as typing judgments. We support typing judgments with three arguments (context, expression, type) and typing judgments with two arguments (expression and type). In our human-readable version of SPL, we represent the first with the classical C |− e : T notation (case class `TypingJudgment`), the later with notation e : T (case class `TypingJudgmentSimple`).

The type system for our running example of typed arithmetic expressions looks like this in SPL:

```
1  data Ty = B | Nat
2
3  ================ T−True
4  true : B
5
6
7  ================ T−False
8  false : B
9
10 ~t1 : B
11 ~t2 : ~T
12 ~t3 : ~T
13 ================ T−If
```

```
14 ifelse(~t1, ~t2, ~t3) : ~T
15
16
17 ================ T−Zero
18 zero : Nat
19
20 ~t1 : Nat
21 ================ T−Succ
22 succ(~t1) : Nat
23
24 ~t1 : Nat
25 ================ T−Pred
26 pred(~t1) : Nat
27
28 ~t1 : Nat
29 ================ T−Iszero
30 iszero(~t1) : B
```

Listing 5.4: Type system of typed arithmetic expressions in SPL

Typing rules may contain *meta variables* (case class `MetaVar`), marked with ~ (tilde) in the code above. Meta variables are implicitly universally quantified. Typing rules T−True, T−False, and T−Zero do not have any premises, hence the empty space above the bar. If a rule has more than one premises, they are separated by a new line (see rule T−If).

We use the same notation as for typing rules also to represent proof goals, lemmmas, axioms, etc. Within the implementation of VeriTaS, we wrap instances of `TypingRule` within case classes `Goals`, `Lemmas`, `Axioms` to mark the difference. For example, we represent the progress property for typed arithmetic expressions as a **goal** as follows:

```
1 goal
2 ~t1 : ~T
3 !isValue(~t1)
4 ============================ Progress
5 exists t2. reduce(~t1) = someTerm(t2)
```

Listing 5.5: Progress property of typed arithmetic expressions in SPL

In Line 2, the exclamation mark ! denotes negation (case class `NotJudgment`). In Line 4, we see how we represent existentially quantified terms (case class `ExistsJudgment`) in SPL. Several existentially quantified variables can be given by separating the variables with a comma before the dot. There is also an analogous construct **forall** for universally quantified terms (case class `ForallJudgment`), which may be used to represent premises within a typing rule with inner universal quantification.

An Embedded DSL for SPL

Using AST case classes for creating concrete specifications using SPL is rather cumbersome and error-prone for users. To provide a more user-friendly input format

for SPL, one may for example generate a parser that processes files written in the format we presented above. This can be done by using parser generators or language workbenches like Spoofax [Vis+14].[1] These options have, however, the disadvantage that users and developers need two different systems to work with VeriTaS: the system in which one may define input specifications and their favorite Scala IDE.

Instead, we provide an embedded DSL within Scala for the SPL AST classes. An embedded DSL has the advantage that users of VeriTaS may work within one environment (namely, their Scala IDE) when developing input specifications and proofs. We provide classes with implicit methods as well as implicit classes to enable a syntax similar to our human-readable version of SPL. These classes and methods internally convert the given terms and symbols to SPL. Note that we cannot use exactly the same syntax as presented previously, since this would sometimes either clash with Scala's keywords or certain symbol chains already defined within Scala. In many places, we need to add elements, keywords, and parentheses in order to give Scala a hint to which implicits apply.

For example, the specification of our running example of typed arithmetic expressions looks as in Listing 5.6 within our embedded DSL for SPL.

```
1  object AEDefs {
2    import DataTypeDSL._
3    import FunctionDSL._
4    import SymTreeDSL._
5    import de.tu_darmstadt.veritas.inputdsl.ProofDSL._
6    import de.tu_darmstadt.veritas.inputdsl.TypingRuleDSL._
7
8    //simple Boolean and arithmetic expressions, syntax
9    val term = data('Term) of
10     'true | 'false | 'ifelse('Term, 'Term, 'Term)
11     'zero | 'succ('Term) | 'pred('Term) | 'iszero('Term)
12
13   ... //omitted some auxiliary predicates
14
15   val getTerm = partial(function('getTerm.>>('OptTerm) -> 'Term) where
16     ('getTerm('someTerm('t)) := 't))
17
18   // reduction semantics
19   val reduce = function('reduce.>>('Term) -> 'OptTerm) where
20     ('reduce('ifelse('true, 't2, 't3)) := 'someTerm('t2)) |
21       ('reduce('ifelse('false, 't2, 't3)) := 'someTerm('t3)) |
22       ('reduce('ifelse('t1, 't2, 't3)) := ((let ('ot1) := 'reduce('t1)) in
23         (iff ('isSomeTerm('ot1))
24           th 'someTerm('ifelse('getTerm('ot1), 't2, 't3))
25           els 'noTerm))) |
26
27    ... //omitted remainder of reduce function
28
```

[1] An earlier version of VeriTaS used an early version of the Spoofax language workbench for this purpose.

```
29  //types (Bool and Nat)
30  val types = data('Ty) of 'B | 'Nat
31
32
33  //typing rules
34  val Ttrue = axiom(
35    ===>("T-true")(
36    'true :: 'B))
37
38  val Tfalse = axiom(
39    ===>("T-false")(
40      'false :: 'B
41    ))
42
43  val Tif = axiom(
44    (('t1 :: 'B) &
45      ('t2 :: 'T) &
46      ('t3 :: 'T)
47      ).===>("T-if")
48      ('ifelse('t1, 't2, 't3) :: 'T))
49
50  ... //omitted remainder of type system specification
51
52  val progress = goal(
53    (('t1 :: 'T) &
54      (!('isValue('t1)))
55      ).===>("Progress")(
56      exists('t2) | ('reduce('t1) === 'someTerm('t2))
57    ))
58
59  ... //omitted other properties
60
61 }
```

Listing 5.6: Typed arithmetic expressions in embedded DSL for SPL

First, in order to be able to use the embedded DSL for SPL, one has to import a number of object methods (line 2 to 6). Importing these object methods allows for using them in specifications without prepending the objects' names. Lines 9 to 11 show how one can define a closed ADT within the embedded DSL. Here we can see the first restriction for terms within the embedded DSL for SPL: For any name, we cannot directly use a Scala variable, since Scala would try to find a definition for this variable term and throw an error when it does not find it. To prevent this issue, we use Scala's type `Symbol` for names. To declare a `Symbol`, Scala allows for simply prepending a ' to the name as syntactic sugar.

Internally, `data` is a method within the object `DataTypeDSL`, which takes a `Symbol` as an argument and creates an intermediate class that contains different versions of a method named `of` (i.e. method `of` is overloaded). These different versions take either a single data type constructor (case class `DataTypeConstructor`), or a list of data type constructors. Object `DataTypeDSL` contains several implicit classes

with methods named |, which construct lists of data type constructors from simple trees of terms formed using `Symbols` (`SymTree` within the code). This allows us to list the different data type constructors separated by | like in our human-readable syntax for SPL when defining closed ADTs.

Lines 15 and 16 show how one can define a partial function within the embedded DSL for SPL. Methods `partial` and `function` are defined within object `FunctionDSL`. Both methods construct appropriate instances of the corresponding AST classes. Often, Scala allows us to omit parentheses around method arguments. However, our method arguments within the embedded DSL for SPL are often so complex that we need to write the parentheses in order to indicate to Scala where a method argument starts and ends. Method `>>` is defined within an implicit class that constructs instances of `FunctionSig` (the case class for function signatures). Similarly, methods `where` and `:=` are defined within implicit classes that construct the corresponding case class instances. We cannot directly use `=` or `:` as method names since these symbols are built into Scala's syntax. Constructing lists of function equations works similarly to constructing lists of data type constructors.

We see a more complex example for a function expression in the third function equation of the reduce function, in lines 22 to 25. Here, we use a let expression and an if expression. Since "let" is not a keyword within Scala, we can define a method `let` within object `FunctionDSL` to construct instances of case class `FunctionExpLet`. For constructing if expressions, we cannot directly use "if", "then", and "else" as syntax, since these three are already keywords within Scala. So we use the slightly different, but similar terms "iff", "th", and "els" instead.

In lines 34 to 36, we see the definition of a simple typing rule within the embedded DSL. Method `===>` is defined within object `TypingRuleDSL` to construct a typing rule without premises. Method `::` is overloaded within different implicit classes in `TypingRuleDSL` that take different arguments. This allows for constructing instances of `TypingJudgmentSimple` within different contexts. In lines 43 to 48, we see a more complex typing rule. Method `&` generates lists of premises. An implicit class takes a list of premises as argument and also contains a method named `===>` that takes the remaining arguments for constructing a typing rule. Prepending symbols with `~` generates an instance of class `MetaVar`. Finally, lines 52 to 57 show how the progress goal looks like within the embedded DSL: Method `goal` generates instances of `Goals`.

Our embedded DSL for SPL is an example for how developers can embed their own input formats into Scala within VeriTaS without great effort: Developers can implement any format for any domain quickly via Scala case classes. Next, they can simply develop an embedded DSL using Scala's implicits as described above to facilitate writing specifications for input problems.

5.1.2. A Subset of Scala for Type System Specifications: ScalaSPL

SPL constitutes a high-level intermediate specification language for VeriTaS, from which we translate into low-level input formats for ATPs and SMT solvers (see

Sections 5.4 and 5.5), similar to how Dafny [Lei10] uses Boogie[2] as an intermediate verification language for translating to SMT-LIB. We designed SPL so that it can be used as an intermediate verification language within different verification domains in the area of programming languages research. Hence, we keep SPL simple and do not extend it with further domain-specific constructs for type soundness proofs.

An input format designed specifically for automating progress and preservation proofs should enable adding domain-specific information to type system specifications which may be used by the automated proof construction.

André Pacak developed the language ScalaSPL as part of his master's thesis [Pac18], which was supervised by the author of the present document. ScalaSPL is a subset of Scala with which one can express specifications of type systems and annotate them with relevant information for the automated generation of progress and preservation proofs. In his master's thesis, Pacak describes in detail which language constructs of Scala ScalaSPL supports, how ScalaSPL is translated into SPL, and the extensible annotation system for ScalaSPL that he developed as well.

ScalaSPL was designed to meet several high-level requirements for an input format for type system specifications, detailed within Pacak's master's thesis [Pac18]. To summarize the main features of ScalaSPL:

- ScalaSPL is a readable format which is natural to Scala users and hence meets the skill of the domain expert we target (see Subsection 1.3.2).
- ScalaSPL specifications are easy to write: Users can exploit all IDE support available for Scala, such as syntax highlighting, code completion, and full type checking.
- ScalaSPL specifications are executable: Since ScalaSPL is a subset of Scala, specifications in Scala may directly be executed. This allows for example for writing tests for ScalaSPL specifications using different Scala test libraries such as ScalaCheck[3]. ScalaSPL specifications may easily be integrated with any other infrastructure available in Scala.
- ScalaSPL features an extensible annotation system for specifying domain-specific information for progress and preservation proofs.

We present ScalaSPL via our running example of typed arithmetic expressions in the following Listing 5.7.

```
1 object AESpec extends ScalaSPLSpecification {
2
3   //simple Boolean and arithmetic expressions
4   sealed trait Term extends Expression
5   case class True() extends Term
6   case class False() extends Term
7   case class Ifelse(b: Term, t: Term, e: Term) extends Term
```

[2] https://boogie-docs.readthedocs.io/en/latest/
[3] https://www.scalacheck.org/

```
8   case class Zero() extends Term
9   case class Succ(p: Term) extends Term
10  case class Pred(s: Term) extends Term
11  case class Iszero(t: Term) extends Term
12
13  def isNV(t: Term): Boolean = t match {
14    case Zero() => true
15    case Succ(nv) => isNV(nv)
16    case _ => false
17  }
18
19  def isValue(t: Term): Boolean = t match {
20    case True() => true
21    case False() => true
22    case t1 => isNV(t1)
23  }
24
25  sealed trait OptTerm
26  case class noTerm() extends OptTerm
27  case class someTerm(t: Term) extends OptTerm
28
29  def isSomeTerm(t: OptTerm): Boolean = t match {
30    case noTerm() => false
31    case someTerm(_) => true
32  }
33
34  @Partial
35  def getTerm(ot: OptTerm): Term = ot match {
36    case someTerm(t) => t
37  }
38
39  //reduction semantics for simple Boolean and arithmetic terms
40  @ProgressProperty("Progress")
41  def reduce(t: Term): OptTerm = t match {
42    case Ifelse(True(), t2, t3) => someTerm(t2)
43    case Ifelse(False(), t2, t3) => someTerm(t3)
44    case Ifelse(t1, t2, t3) =>
45      val ot1 = reduce(t1)
46      if (isSomeTerm(ot1))
47        someTerm(Ifelse(getTerm(ot1), t2, t3))
48      else
49        noTerm()
50    case Succ(t1) =>
51      val ot2 = reduce(t1)
52      if (isSomeTerm(ot2))
53        someTerm(Succ(getTerm(ot2)))
54      else
55        noTerm()
56    case Pred(Zero()) => someTerm(Zero())
57    case Pred(Succ(nv)) =>
58      if (isNV(nv)) someTerm(nv)
```

```
59        else { val ot2 = reduce(Succ(nv))
60          if (isSomeTerm(ot2))
61            someTerm(Pred(getTerm(ot2)))
62          else
63            noTerm()
64          }
65      case Pred(t1) =>
66        val ot2 = reduce(t1)
67        if (isSomeTerm(ot2))
68          someTerm(Pred(getTerm(ot2)))
69        else
70          noTerm()
71      case Iszero(Zero()) => someTerm(True())
72      case Iszero(Succ(nv)) =>
73        if (isNV(nv)) someTerm(False())
74        else { val ot2 = reduce(Succ(nv))
75          if (isSomeTerm(ot2))
76            someTerm(Iszero(getTerm(ot2)))
77          else
78            noTerm()
79          }
80      case Iszero(t1) =>
81        val ot2 = reduce(t1)
82        if (isSomeTerm(ot2))
83          someTerm(Iszero(getTerm(ot2)))
84        else
85          noTerm()
86      case _ => noTerm()
87    }
88
89    //types (Bool and Nat)
90    sealed trait Ty extends Type
91    case class B() extends Ty
92    case class Nat() extends Ty
93
94    //typing rules
95    @Axiom
96    def Ttrue(): Unit = {} ensuring (True() :: B())
97
98    @Axiom
99    def Tfalse(): Unit = {} ensuring(False() :: B())
100
101   @Axiom
102   def Tif(t1: Term, t2: Term, t3: Term, T: Ty): Unit = {
103     require(t1 :: B())
104     require(t2 :: T)
105     require(t3 :: T)
106   } ensuring(Ifelse(t1, t2, t3) :: T)
107
108   @Axiom
109   def TNat(): Unit = {} ensuring(Zero() :: Nat())
```

```
110
111   @Axiom
112   def TSucc(t1: Term): Unit = {
113     require(t1 :: Nat())
114   } ensuring(Succ(t1) :: Nat())
115
116   @Axiom
117   def TPred(t1: Term): Unit = {
118     require(t1 :: Nat())
119   } ensuring(Pred(t1) :: Nat())
120
121   @Axiom
122   def Tiszero(t1: Term): Unit = {
123     require(t1 :: Nat())
124   } ensuring(Iszero(t1) :: B())
125
126
127   // steps for soundness proof (progress and preservation) for typed arithmetic expressions
          as given in Pierce, TAPL, Chapter 8
128   @Property
129   def Progress(t1: Term, T: Ty): Unit = {
130     require(t1 :: T)
131     require(!isValue(t1))
132   } ensuring exists( (t2: Term) => reduce(t1) == someTerm(t2))
133
134 }
```

Listing 5.7: Typed arithmetic expressions in ScalaSPL

As we can see for example in lines 4 to 11, ScalaSPL uses Scala's `sealed trait`s for modeling closed algebraic datatypes. Normal traits, without keyword `sealed`, model SPL's open datatypes. Scala's case classes which extend a trait or sealed trait express individual data type constructors.

All ScalaSPL specifications have to extend trait `ScalaSPLSpecification` (see line 1), which provides ScalaSPL's basic syntactic constructs and certain domain-specific constructs for specifications of type systems. Among these are the inner base traits `Expression` (line 4) and `Type` (line 76), which are used for providing syntactic sugar for typing judgments (e.g. in line 95, `Zero() :: Nat()`): Users of ScalaSPL have to mark expressions of a language with trait `Expression` and types of a language with trait `Type` to use this syntactic sugar.

Function definitions within ScalaSPL constitute of normal definitions (via `def`) in Scala, but have to consist of a `match` expression, whose individual cases specify the function's behavior for the given pattern (see for example lines 13-16). Each case expression is translated to a separate function equation in SPL.

The individual cases may contain function applications or case class instances of the function's return type, variable definitions via `val` or if-else expressions (see for example lines 45 to 49). Variable definitions become let-expressions in SPL.

The lines following line 80 show how one can define inference rules (used for specifying typing rules as well as properties, just like in SPL) in ScalaSPL: via

definitions with return type `Unit` and an empty body. ScalaSPL uses Scala's `require`-Notation for premises and the `ensuring`-notation for conclusions of inference rules. Note that we have to declare all implicitly universally quantified variables within an inference rule within the function signature, otherwise we would not obtain valid Scala code.

The previous ScalaSPL listing contains some of the annotations available in ScalaSPL. For example, lines 81, 84, 87 etc. mark the typing rules of typed arithmetic expressions as axioms, line 114 marks the progress property as a property. Line 40 shows a domain-specific annotation: The annotation `@ProgressProperty(name)` allows for declaring which property (via its `name`) of the progress pattern belongs to the function with that annotation. Chapter 6 goes into more details regarding the available domain-specific annotation in ScalaSPL and how they are used for automated proof construction.

Users may use ScalaSPL in VeriTaS as follows: They create a ScalaSPL specification and use the provided translator to translate it to SPL. Then, they create concrete proof graph instances, instantiating the specification and goal format with `VeritasConstruct` and `VeritasFormula`. For growing proof graphs, they use the basic tactics for SPL that we will describe next. They may inspect nodes of the proof graph in ScalaSPL by invoking a provided pretty-printer that prints SPL specifications as ScalaSPL code.

5.2. Basic Tactics

We implemented standard proof tactics with to prove simple properties given in SPL: structural induction, case distinctions, and lemma applications. These tactics turn out to be sufficient for our target verification domain.

Each tactic receives tactic-specific information from the specification, the parent obligation on which it is applied along with the incoming proof edges of the parent obligation, and an obligation producer. The tactics all use the given obligation producer to create new obligations for a proof graph. The incoming proof edges may contain information that needs to be propagated along the proof graph, such as induction hypotheses and fixed variables. Ultimately, the tactics generate a collection of new sub-obligations and associated proof edges that are to be added to the proof graph.

When implementing tactics in our verification infrastructure, the usage of a modern object-oriented general-purpose programming language, namely Scala, allows us to employ any kind of advantageous software engineering techniques, for example in order to increase the reusability of the implemented tactics.

To illustrate this approach, we first defined a lightweight trait `SpecEnquirer` for querying specific information from problem specifications. The trait is, like all the other parts of our VeriTaS core API described in Section 4.3, parametric in a format for problem specifications and proof obligations. Trait `SpecEnquirer` contains methods for querying basic information from a specification format, such as for example "Is a given proof obligation universally quantified?", "Does a given

variable in a given term have the type of a closed ADT?" etc. Additionally, trait `SpecEnquirer` contains constructor methods for building proof obligations.

Next, we implemented our basic tactics as generic tactics that are again parametric in a format for problem specifications (`Def`) and in a format for proof obligations (`Goal`). The implemented tactics make no assumptions about how a specification format looks like, but employ solely the query methods from trait `SpecEnquirer` to obtain relevant information from parts of the given specification and the constructor methods of `SpecEnquirer` to build sub-obligations and proof edges. Finally, we implemented trait `SpecEnquirer` for SPL.

This design allows the tactics that we implemented to be reused with other specification formats beyond SPL. To reuse the tactics with a custom format, developers only have to implement the `SpecEnquirer` trait for their own format.

Note that, in accordance with our requirement 4.3 of decoupling proof construction and step verification, tactic applications on proof graphs need not necessarily represent correct proof steps. A tactic only has to create a proof problem that can be passed to an external verifier, which then has to attempt the actual verification of the proof step. For example, the tactic for structural induction creates base cases, step cases, and induction hypotheses for a given proof obligation and a given induction variable, based on the type of this variable. The associated proof step consists of the induction cases and hypotheses and the parent obligation. To verify such a step, a verifier has to confirm that the generated induction cases conform to a valid induction scheme.

We summarize how each of our basic tactics operates exactly:

- **Structural induction:** The structural induction tactic receives an induction variable on which to apply structural induction. It uses a given `SpecEnquirer` to obtain the data-type constructors associated to the type of the given induction variable (if applicable). Next, it uses again the given `SpecEnquirer` to create one sub-obligation per constructor as well as induction hypotheses for recursive constructor arguments. For creating the new sub-obligations, the tactic simply adds a new premise to the parent goal with an equation that fixes the induction variable to be equal a constructor term in question. When generating induction hypotheses, the tactic also generates fixed variables, i.e., the variables that appear in recursive positions within an induction case and are used within the induction hypotheses. The tactic generates new proof edges, including propagatable information from the given parent edges and the newly created induction hypotheses and fixed variables. Finally, the tactic returns the newly created obligations for the induction cases and the corresponding proof edges. The tactic fails if the given induction variable does not exist within the parent obligation or does not have a closed algebraic datatype as type.

- **Case distinctions:** We implement different case distinction tactics (for general case distinctions, structural case distinctions, and boolean case distinctions). The general case distinction tactic receives a list of case predicates. It generates one new sub-obligation for each predicate by simply adding the

predicate as additional premise to the parent goal. For the generation of these sub-obligation, it uses methods from the `SpecEnquirer`. The other case distinction tactics are specializations of the general case distinction tactic: The structural case distinction tactic only receives a variable which is of a closed algebraic datatype and generates one case predicate per constructor of this datatype. The boolean case distinction receives a single predicate and generates two sub-obligations, one for the predicate and one for the negation of the predicate. All case distinction tactics propagate the information from parent proof edges to the proof edges they generate for each new case.

- **Lemma application:** The lemma application tactic receives a list of lemmas. It simply creates one new proof obligation for each lemma. This tactic does not propagate any information along the newly generated proof edges: For each lemma, we will start a new proof, where induction hypotheses and fixed variables steps further up in the proof graph cannot be available anymore.

5.3. Connecting Different Verifiers

To connect different ATPs and SMT solvers to VeriTaS, we need to

1. implement a translation from the specification format for definitions and proof obligations that we use to an input format supported by the prover/solver we want to connect,
2. parse the output or logs of the prover and translate them into the `StepResult` used by the `ProofGraph` instantiation that we use,
3. implement the `Verifier` trait with a verifier that calls the external prover with the translated problem and parses its output.

We implemented different translations from our custom specification DSL into different TPTP dialects [Sut17], which is supported by many different ATPs, and also into the SMT-LIB format [BFT16], which allows us to connect SMT solvers such as Z3 [DB08]. We describe our translations to these formats in detail in the following sections. Finally, we implemented `Verifier`s which call different versions of Vampire [KV13], and `Verifier`s which call Eprover [Sch13], and princess [Rüm08b]. We parse the results of these provers into `StepResults`. Vampire hands back TSTP [Sut10] proofs, which we use as prover evidence. We also implemented a verifier that calls Z3 [DB08]. All of the verifiers we implement require that users have appropriate binaries of them installed and available from their PATH.

5.4. Encoding SPL in TPTP

We describe how we translate SPL into TPTP [Sut10]. This translation can be seen as a compilation (in general, compilers simply translate from one programming language into another). TPTP contains different "dialects", e.g. "fof" for classical

untyped first-order logic and "tff" for *typed* first-order logic. We describe first the compilation strategy to *typed* first-order logic, this being a little closer to SPL, which is also typed.

TPTP is a machine-readable format. To increase the readability of the section below, we represent all formula schemes and examples in the standard typed FOL-notation as introduced in Section 2.1. Translating from that notation to TPTP is straightforward and just involves taking into account some uninteresting minor technical details (e.g. all universally quantified variables in TPTP have to start with a lower-case letter, hence we simple prepend "v" to every variable name to make sure this is indeed the case in the final TPTP output).

5.4.1. Encoding Data Types

To encode closed algebraic data types of the form **data** $\mathsf{N} = \mathsf{c}_1(\overline{\mathsf{T}_1}) \ldots \mathsf{c}_n(\overline{\mathsf{T}_n})$ in typed first-order logic, we first generate a function symbol $\mathsf{c}_i : \overline{\mathsf{T}_i} \to \mathsf{N}$ for each constructor. Second, we generate the following axioms to specify the algebraic nature of SPL data types:

1. Constructor functions are *injective*:

$$\bigwedge_{k \in \{1..n\}} (\forall \overline{x}, \overline{y}.\ \ c_k(\overline{x}) = c_k(\overline{y}) \Longrightarrow \bigwedge_i x_i = y_i)$$

2. Calls to *different constructors* always yield distinct results:

$$\bigwedge_{i \neq j} \forall \overline{x_i}, \overline{x_j}.\ \ c_i(\overline{x_i}) \neq c_j(\overline{x_j})$$

3. Each term of data type N must be of a constructor form. We call the resulting axiom the *domain axiom* for data type N:

$$\forall t{:}\mathsf{N}.\quad \bigvee_i \exists \overline{x_i}.\ t = c_i(\overline{x_i})$$

For example, for data type Term from Listing 5.2, we generate the following function symbols and axioms in typed first-order logic:

```
1  true: Term
2  false: Term
3  zero: Term
4  succ: Term → Term
5  pred: Term → Term
6  iszero: Term → Term
7  ifelse: Term × Term × Term → Term
8
9  ∀ t1:Term, t2:Term. succ(t1) = succ(t2) ⟹ t1 = t2
10 ∀ t1:Term, t2:Term. pred(t1) = pred(t2) ⟹ t1 = t2
11 ∀ t1:Term, t2:Term. iszero(t1) = iszero(t2) ⟹ t1 = t2
12 ∀ t1:Term, t2:Term, t3:Term, s1:Term, s2:Term, s3:Term. ifelse(t1, t2, t3) = ifelse(s1, s2, s3)
       ⟹ (t1 = s1 ∧ t2 = s2 ∧ t3 = s3)
13
14 true ≠ false
15 true ≠ zero
```

```
16 ∀ t1:Term. true ≠ succ(t1)
17 ∀ t1:Term. true ≠ pred(t1)
18 ∀ t1:Term. true ≠ iszero(t1)
19 ∀ t1:Term, t2:Term, t3:Term. true ≠ ifelse(t1, t2, t3)
20 false ≠ zero
21 ∀ t1:Term. false ≠ succ(t1)
22 ∀ t1:Term. false ≠ pred(t1)
23 ∀ t1:Term. false ≠ iszero(t1)
24 ∀ t1:Term, t2:Term, t3:Term. false ≠ ifelse(t1, t2, t3)
25 ∀ t1:Term. zero ≠ succ(t1)
26 ∀ t1:Term. zero ≠ pred(t1)
27 ∀ t1:Term. zero ≠ iszero(t1)
28 ∀ t1:Term, t2:Term, t3:Term. zero ≠ ifelse(t1, t2, t3)
29 ∀ t1:Term, t2:Term. succ(t1) ≠ pred(t2)
30 ∀ t1:Term, t2:Term. succ(t1) ≠ iszero(t2)
31 ∀ t1:Term, t2:Term, t3:Term, t4:Term. succ(t1) ≠ ifelse(t2, t3, t4)
32 ∀ t1:Term, t2:Term. pred(t1) ≠ iszero(t2)
33 ∀ t1:Term, t2:Term, t3:Term, t4:Term. pred(t1) ≠ ifelse(t2, t3, t4)
34 ∀ t1:Term, t2:Term, t3:Term, t4:Term. iszero(t1) ≠ ifelse(t2, t3, t4)
35
36 ∀ t:Term. t = true ∨ t = false ∨ t = zero ∨ ∃ t1:Term. t = succ(t1) ∨ ∃ t1:Term. t =
       pred(t1) ∨ ∃ t1:Term. t = iszero(t1) ∨ ∃ t1:Term, t2:Term, t3:Term. t = ifelse(t1, t2, t3)
```

Listing 5.8: Axioms for datatype Term in typed first-order logic

Lines 1 to 7 show which function symbols are generated, where true, false, and zero are functions with zero arguments. Lines 9 to 12 show the injectivity axioms for datatype Term, which ensure that for example two terms of the form succ(...) are always considered equal if their constructor arguments are equal. Lines 14 to 34 show all the difference axioms (all pairwise combinations between the seven constructors of Term), which ensure that terms which start with different constructors are never treated as equal. Finally, line 36 shows the domain axiom for terms of type Term, which ensures that type Term is indeed closed.

For an open data type N, we generate an axiomatization that ensures N is countably infinite as desired:

```
1 init_N : N
2 enum_N : N → N
3 ∀ x1:N, x2:N. x1 ≠ x2 ⟹ enum_N(x1) ≠ enum_N(x2)
4 ∀ x:N. init_N ≠ enum_N(x)
```

Listing 5.9: Axiom schemata for open datatypes

Intuitively, these axioms define that the structure of an open data type N is isomorphic to the structure of natural numbers, which are also countably infinite ($init_N$ corresponds to the initial element zero of natural numbers, $enum_N$ to the successor function).

Finally, we directly translate constant symbols const x:T to function symbols with zero arguments x:T in typed first-order logic.

5.4.2. Encoding Function Specifications

We encode partial and total SPL functions of the form

```
1 (partial) function f : T_1 ... T_n → T
2 f(p_1) = e_1
3 ...
4 f(p_m) = e_m
```

Listing 5.10: Scheme for functions in SPL

in first-order logic by axiomatizing the equations. In the given scheme for SPL function above, the notation $\overline{\mathsf{p_i}}$ abbreviates $\mathsf{p_{i,1}}, ..., \mathsf{p_{i,n}}$, i.e. the n argument patterns in function equation i.

We apply four translation steps to subsequently eliminate conditionals, **let**-bindings, equation ordering, and free variables from the SPL function equations. This way, we produce increasingly refined formulas $\phi^{\mathsf{k}}_{\mathsf{i}}$ for function equation i after translation step k.

1. Conditionals: For each **if**-expression in a function equation i of the form $\mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{C_i}[\textbf{if } \mathsf{c\ t\ e}]$ for some context $\mathsf{C_i}$, we split equation i in two to handle positive and negative cases separately:

$$\begin{array}{lcl} \phi^1_{\mathsf{i,c}} & := & \mathsf{c} \implies \mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{C_i}[\mathsf{t}] \\ \phi^1_{\mathsf{i},\neg\mathsf{c}} & := & \neg\mathsf{c} \implies \mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{C_i}[\mathsf{e}] \end{array}$$

   In the notation above, we add the condition c and its negation $\neg\mathsf{c}$ as subscripts to $\phi^{\mathsf{k}}_{\mathsf{i}}$ to differentiate the different formulae that result for function equation i.
2. Bindings: For each **let**-binding in a function equation i of the form $\mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{C_i}[\textbf{let } \mathsf{x\ a\ e}]$ for some context $\mathsf{C_i}$, we add a precondition representing the binding to the preconditions $\mathsf{pc}^1_{\mathsf{b}}(\mathsf{i})$ produced in step 1, where subscript b represents the conjunction of Boolean conditions generated in step 1:

$$\phi^2_{\mathsf{i,b}} := \mathsf{pc}^1_{\mathsf{b}}(\mathsf{i}) \wedge\ \mathsf{x} = \mathsf{a} \implies \mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{C_i}[\mathsf{e}]$$

   When adding preconditions variable bindings, we also ensure scope preservation for **let**-bound variables, renaming variables where necessary.
3. Equation order: This step encodes the equation order from the original SPL specification to typed first-order logic, where the order of axioms is irrelevant. Our encoding ensures that at most one function equation is applicable for a given argument pattern no matter how the axioms are ordered. For each function equation i of the form $\mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{e_i}$, we add inequalities NPC that exclude all function patterns $\overline{\mathsf{p_j}}$ from previously seen equations $\mathsf{j} < \mathsf{i}$:

$$\begin{array}{l} \mathsf{NPC}(\mathsf{i}) := \bigwedge_{\mathsf{j}<\mathsf{i}} \overline{\mathsf{p_i}}\ \neq\ \overline{\mathsf{p_j}} \\ \phi^3_{\mathsf{i,b}} := \mathsf{pc}^2_{\mathsf{b}}(\mathsf{i}) \wedge \mathsf{NPC}(\mathsf{i}) \implies \mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{e_i} \end{array}$$

   The function NPC generates a disjunction of inequalities which ensure that only terms fitting function pattern $\mathsf{p_i}$ are considered if the patterns from previous function equations do not match for them. Function NPC ensures that variable

names in $\overline{\mathsf{p_i}}$ and in $\overline{\mathsf{p_j}}$ do not clash. $\mathsf{pc}^2_\mathsf{b}(\mathsf{i})$ represents all preconditions added after step 1 and 2 for equation i.

4. Quantify free variables: We close each formula by universally quantifying over the variables $\overline{\mathsf{a_i}}$ with their corresponding types in function patterns $\overline{\mathsf{p_i}}$ and over all other free variables $\overline{\mathsf{x_i}}$ with their corresponding types that appear in $\phi^3_{\mathsf{i,b}}$ (such as the ones introduced for **let**-bindings).

$$\phi^4_{\mathsf{i,b}} := \forall \overline{\mathsf{a_i}}.\forall \overline{\mathsf{x_i}}.\ \phi^3_{\mathsf{i,b}}$$

Above, the notation $\overline{\mathsf{a_i}}$ abbreviates $\mathsf{a}_{\mathsf{i},1} : \mathsf{A}_{\mathsf{i},1}, ..., \mathsf{a}_{\mathsf{i},\mathsf{k}} : \mathsf{A}_{\mathsf{i},\mathsf{k}}$, i.e. all k variables appearing in function pattern $\mathsf{p_i}$ with their corresponding types $\mathsf{A}_{\mathsf{i},1}$ to $\mathsf{A}_{\mathsf{i},\mathsf{k}}$.

For functions that return Boolean values, after translation, we replace equations $\mathsf{f}(\overline{\mathsf{p_i}}) = \mathsf{e_i}$ by biimplications $\mathsf{f}(\overline{\mathsf{p_i}}) \iff \mathsf{e_i}$. This step is necessary since our target format TPTP [Sut10] does not allow Boolean values as arguments of equalities or inequalities. A corresponding language extension to TPTP that allows Boolean values as arguments is developed in [Kot+16b], but, to the author's current knowledge, not yet supported by all theorem provers that we connected to VeriTaS.

As an example for how we axiomatize SPL functions, we show an excerpt of the axiomatization of the function reduce from Listing 5.3.

```
1  reduce: Term → OptTerm
2  ∀ t2:Term, t3:Term. reduce(ifelse(true, t2, t3)) = someTerm(t2)
3  ∀ t2:Term, t3:Term. ifelse(false, t2, t3) ≠ ifelse(true, t2, t3) ∧
4    reduce(ifelse(false, t2, t3)) = someTerm(t3)
5  ∀ t1:Term, t2:Term, t3:Term. ∀ ot1:OptTerm. isSomeTerm(ot1) ∧ ot1 = reduce(t1) ∧
6    ifelse(t1, t2, t3) ≠ ifelse(true, t2, t3) ∧ ifelse(t1, t2, t3) ≠ ifelse(false, t2, t3)
7        ⟹ reduce(ifelse(t1, t2, t3)) = someTerm(ifelse(getTerm(ot1), t2, t3))
8  ∀ t1:Term, t2:Term, t3:Term. ∀ ot1:OptTerm. ¬isSomeTerm(ot1) ∧ ot1 = reduce(t1) ∧
9    ifelse(t1, t2, t3) ≠ ifelse(true, t2, t3) ∧ ifelse(t1, t2, t3) ≠ ifelse(false, t2, t3)
10       ⟹ reduce(ifelse(t1, t2, t3)) = noTerm
11 ...
12 ∀ t:Term. ∀ t:1:Term, t2:Term, t3:Term, t4:Term, t5:Term, t6:Term.
13   t ≠ ifelse(true, t2, t3) ∧ t ≠ ifelse(false, t2, t3) ∧ t ≠ ifelse(t1, t2, t3) ∧ t ≠ succ(t1) ∧
14   t ≠ pred(zero) ∧ t ≠ pred(succ(nv)) ∧ t ≠ pred(t1) ∧ t ≠ iszero(zero) ∧
15   t ≠ iszero(succ(nv)) ∧ t ≠ iszero(t1)
16       ⟹ reduce(t) = noTerm
```

Listing 5.11: Axiomatization in typed first-order logic of function reduce from Listing 5.3 (excerpt)

The first two equations (lines 2 and 3 of Listing 5.11) represent the axiomatization of the first two function equations of the reduce function from Listing 5.3 (lines 13 and 14). The equations are encoded almost "as is", only quantifying the free variables and adding a pattern inequality for the second function equation. The third and fourth axiom in Listing 5.11 (lines 4 to 9) encode the third function equation in Listing 5.3 (lines 15 to 19): This function equation contains an **if**-expression, hence we generate two axioms for each alternative. The first axiom contains a premise for the positive condition of the if-expression (isSomeTerm(ot_1)), the second

one a premise for the negative condition ($\neg\mathsf{isSomeTerm}(\mathsf{ot}_1)$). Both axioms encode the **let**-binding via an equational premise ($\mathsf{ot}_1 = \mathsf{reduce}(\mathsf{t}_1)$). We add inequalities as premises to distinguish the argument pattern of the current function equation ($\mathsf{ifelse}(\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3)$) from argument patterns in the previous two function equations. Note that when inlining further knowledge about our encoding of ADTs to typed first-order logic, some of the pattern inequalities above can be further simplified. We discuss these simplifications in Subsection 8.1.3.

For each total function, we generate an *inversion axiom* to encode the inversion property of total functions. The inversion axiom states that a total function is fully defined by its equations and that at least one of the equations must hold. Conversely, the preconditions in $\mathsf{pc}^3_\mathsf{b}(\mathsf{i})$ introduced via NPC ensure that at most one equation can hold for any given argument pattern $\overline{\mathsf{p_i}}$. This way our encoding retains the determinism of functions. In our initial experiments, we observed that inversion axioms are not always needed, but often seem to help ATPs to prove the goals we investigate.

Concretely, the inversion axiom for the function equation axioms of the form in Listing 5.12 takes the form given in Listing 5.13.

$\mathsf{f} : \mathsf{T}_1 \ ... \ \mathsf{T}_\mathsf{n} \to \mathsf{T}$
$\phi^4_{1,\mathsf{b}} := \forall\overline{\mathsf{a_1}}.\forall\overline{\mathsf{x_1}}.\mathsf{pc}^3_\mathsf{b}(1) \implies \mathsf{f}(\overline{\mathsf{p_1}}) = \mathsf{e}_1$
$\phi^4_{1,\mathsf{c}} := \forall\overline{\mathsf{a_1}}.\forall\overline{\mathsf{x_1}}.\mathsf{pc}^3_\mathsf{c}(1) \implies \mathsf{f}(\overline{\mathsf{p_1}}) = \mathsf{f}_1$
...
$\phi^4_{\mathsf{m},\mathsf{z}} := \forall\overline{\mathsf{a_m}}.\forall\overline{\mathsf{x_m}}.\mathsf{pc}^3_\mathsf{z}(\mathsf{m}) \implies \mathsf{f}(\overline{\mathsf{p_m}}) = \mathsf{e}_\mathsf{m}$

Listing 5.12: Axiom scheme of encoded function equations

1 $\forall \mathsf{t}_1 : \mathsf{T}_1, \ ..., \ \mathsf{t}_\mathsf{n} : \mathsf{T}_\mathsf{n}.$
2 $\quad(\exists\overline{\mathsf{a_1}}.\exists\overline{\mathsf{x_1}}.(\bigwedge_{\mathsf{i}\in\{1..\mathsf{n}\}} \mathsf{t}_\mathsf{i} = \mathsf{p}_{1,\mathsf{i}}) \wedge \mathsf{pc}^3_\mathsf{b}(1) \wedge \mathsf{f}(\mathsf{t}_1, ..., \mathsf{t}_\mathsf{n}) = \mathsf{e}_1)$
3 $\quad\vee(\exists\overline{\mathsf{a_1}}.\exists\overline{\mathsf{x_1}}.(\bigwedge_{\mathsf{i}\in\{1..\mathsf{n}\}} \mathsf{t}_\mathsf{i} = \mathsf{p}_{1,\mathsf{i}}) \wedge \mathsf{pc}^3_\mathsf{c}(1) \wedge \mathsf{f}(\mathsf{t}_1, ..., \mathsf{t}_\mathsf{n}) = \mathsf{f}_1)$
4 $\quad\vee ...$
5 $\quad\vee(\exists\overline{\mathsf{a_m}}.\exists\overline{\mathsf{x_m}}.(\bigwedge_{\mathsf{i}\in\{1..\mathsf{n}\}} \mathsf{t}_\mathsf{i} = \mathsf{p}_{\mathsf{m},\mathsf{i}}) \wedge \mathsf{pc}^3_\mathsf{z}(\mathsf{m}) \wedge \mathsf{f}(\mathsf{t}_1, ..., \mathsf{t}_\mathsf{n}) = \mathsf{e}_\mathsf{m})$

Listing 5.13: Scheme of generated inversion axioms

That is, we introduce fresh variables $\mathsf{t}_1 : \mathsf{T}_1, \ ..., \ \mathsf{t}_\mathsf{n} : \mathsf{T}_\mathsf{n}$ (line 1 in Listing 5.13) and then create a conjunction with as many arguments as generated function axioms. For each argument of the big conjunction, we turn the universal quantification of $\overline{\mathsf{a_i}}$ and $\overline{\mathsf{x_i}}$ into existential quantifications and append a disjunction of equations that fix the function's pattern ($\bigwedge_{\mathsf{i}\in\{1..\mathsf{n}\}} \mathsf{t}_\mathsf{i} = \mathsf{p}_{1,\mathsf{i}}$), the preconditions $\mathsf{pc}^3_\mathsf{b}(\mathsf{i})$ of the corresponding $\phi^4_{\mathsf{i},\mathsf{b}}$, and the final function equation from $\phi^4_{\mathsf{i},\mathsf{b}}$ where the pattern variables are replaced with the freshly introduced variables $\mathsf{t}_1 : \mathsf{T}_1, ..., \mathsf{t}_\mathsf{n} : \mathsf{T}_\mathsf{n}$.

As an example for an inversion lemma, we show an excerpt of the inversion lemma for the reduce function whose axiomatization we saw in Listing 5.11:

1 $\forall$ t_0: Term.
2 $\quad(\exists$ t_2: Term, t_3: Term. $\mathsf{t}_0 = \mathsf{ifelse}(\mathsf{true}, \mathsf{t}_2, \mathsf{t}_3) \wedge \mathsf{reduce}(\mathsf{t}_0) = \mathsf{someTerm}(\mathsf{t}_2))$
3 $\quad\vee\ (\exists$ t_2: Term, t_3: Term. $\mathsf{t}_0 = \mathsf{ifelse}(\mathsf{false}, \mathsf{t}_2, \mathsf{t}_3) \wedge \mathsf{ifelse}(\mathsf{false}, \mathsf{t}_2, \mathsf{t}_3) \neq \mathsf{ifelse}(\mathsf{true}, \mathsf{t}_2, \mathsf{t}_3)$
4 $\quad\quad\wedge\ \mathsf{reduce}(\mathsf{t}_0) = \mathsf{someTerm}(\mathsf{t}_2))$

```
5   ∨ (∃ t1: Term, t2: Term, t3: Term. ∃ ot1:OptTerm. t0 = ifelse(t1, t2, t3)
6     ∧ isSomeTerm(ot1) ∧ ot1 = reduce(t1)
7     ∧ ifelse(t1, t2, t3) ≠ ifelse(true, t2, t3) ∧ ifelse(t1, t2, t3) ≠ ifelse(false, t2, t3)
8       ∧ reduce(t0) = someTerm(ifelse(getTerm(ot1), t2, t3))
9   ∨ (∃ t1: Term, t2: Term, t3: Term. ∃ ot1:OptTerm. t0 = ifelse(t1, t2, t3)
10    ∧ ¬isSomeTerm(ot1) ∧ ot1 = reduce(t1)
11    ∧ ifelse(t1, t2, t3) ≠ ifelse(true, t2, t3) ∧ ifelse(t1, t2, t3) ≠ ifelse(false, t2, t3)
12      ∧ reduce(t0) = noTerm))
13  ...
14  ∨ (∃ t:Term. ∃ t:1:Term, t2:Term, t3:Term, t4:Term, t5:Term, t6:Term. t0 = t ∧
15    t ≠ ifelse(true, t2, t3) ∧ t ≠ ifelse(false, t2, t3) ∧ t ≠ ifelse(t1, t2, t3) ∧ t ≠ succ(t1) ∧
16    t ≠ pred(zero) ∧ t ≠ pred(succ(nv)) ∧ t ≠ pred(t1) ∧ t ≠ iszero(zero) ∧
17    t ≠ iszero(succ(nv)) ∧ t ≠ iszero(t1)
18      ∧ reduce(t0) = noTerm)
```

Listing 5.14: Inversion lemma for reduce function from Listing 5.3 (excerpt)

For functions with Boolean result type, we generate two inversion lemmas: one that describes all possible conditions for pairs of function arguments and return expressions if the function returns true, and one that describes all possible conditions if the function returns false. Both lemmas take the form as presented in Listing 5.13, only that the equations $\mathsf{f}(\mathsf{t_1}, ..., \mathsf{t_n}) = \mathsf{e_i}$ are replaced by the Boolean conditions $\mathsf{f}(\mathsf{t_1}, ..., \mathsf{t_n})$ respectively $\neg\mathsf{f}(\mathsf{t_1}, ..., \mathsf{t_n})$.

Since functions in FOL are always total, we deliberately do not generate inversion axioms for partial functions. Thus, we prevent the prover from reasoning about a valid argument/result pair for an argument for which the corresponding partial function is not defined in the original SPL specification.

5.4.3. Encoding Inference Rules and Properties

The encoding of inference rules resp. properties in SPL to FOL is straightforward, since the individual premises and conclusions of these rules are already in FOL. The SPL meta-variables (the variables marked with ˜) in inference rules become normal variables in typed FOL. To encode typing judgments of the form $\mathsf{C} \mid\!- \mathsf{e} : \mathsf{T}$ resp. $\mathsf{e} : \mathsf{T}$, which may appear within premises or conclusions of inference rules resp. properties in SPL, we first generate appropriate function predicates $\mathsf{tpcheck} : \mathsf{C_T}\ \mathsf{e_T}\ \mathsf{T_T} \to \mathsf{Bool}$ resp. $\mathsf{tpchecksimple} : \mathsf{e_T}\ \mathsf{T_T} \to \mathsf{Bool}$. In the previous sentence, $\mathsf{C_T}$ is the type of C, $\mathsf{e_T}$ is the type of e, and $\mathsf{T_T}$ is the type of T, which we infer from the specification using a standard type inference algorithm. Next, we replace all occasions of $\mathsf{C} \mid\!- \mathsf{e} : \mathsf{T}$ resp. $\mathsf{e} : \mathsf{T}$ within premises and conclusions with applications of the generated function predicates. Finally, after translating premises and conclusions, we encode inference rules with premises $\mathsf{pre_i}$ and conclusions $\mathsf{con_j}$ simply as implications $\forall\overline{\mathsf{mv}}.(\bigwedge_\mathsf{i} \mathsf{pre_i}) \implies (\bigwedge_\mathsf{j} \mathsf{con_j})$, where $\overline{\mathsf{mv}}$ represents the list of all SPL meta-variables in the inference rule together with their types (which we also infer during the translation). We mark typing rules and axioms as *axioms* and properties as *conjectures* within FOL.

For example, we encode the typing rule **T-If** from typed arithmetic expressions (lines 10 to 14 in Listing 5.4) as the following axiom in FOL:

```
1  ∀ t1: Term, t2: Term, t3: Term, T: Ty.
2    tpchecksimple(t1, B) ∧ tpchecksimple(t2, T) ∧ tpchecksimple(t3, T)
3      ⟹ tpchecksimple(ifelse(t1, t2, t3), T)
```

Listing 5.15: Typing rule **T-If** as axiom in FOL

5.5. Encoding SPL in SMT-LIB

Another standard format within the automated theorem prover community is SMT-LIB [BFT16], which is traditionally used by SMT solvers. Encoding proof problems in SMT-LIB allows for using SMT solvers such as Z3 [DB08] as well as ATPs with special features such as Vampire with support for term-algebraic reasoning [KRV17] to solve proof steps in VeriTaS.

The SMT-LIB format is conceptionally very close to SPL, the core specification language in VeriTaS (see Section 5.1.1): It features constructs for closed ADTs and for specifications of recursive functions via function equations. We implemented the translation to SMT-LIB by reusing a great part of the compiler product line we described in Subsection 8.1.4. We slightly modified existing transformation steps and assembled them to a custom compilation strategy. Then, we replaced the last step of the chain (where low-level SPL is translated into TPTP) with a simple syntactic translation to SMT-LIB. Thus, the translation to SMT-LIB demonstrates the reusability of our modular compiler product line that originally compiles SPL to TPTP.

Below, we briefly elaborate by example how the individual SPL language constructs are eventually encoded into SMT-LIB.

5.5.1. Encoding Data Types

SMT-LIB has its own construct for closed ADT, called declare−datatypes. The declare−datatypes construct supports more information than the simple SPL ADTs: It supports parametric datatypes and requires specifying selector functions for all constructor arguments. Hence, when encoding ADTs from SPL in SMT-LIB, we need to additionally specify that a datatype has no type parameters (since type parameters are not supported by SPL) and we need to generate names for selector functions.

For example, encoding the datatype Term from our running example of typed arithmetic expressions (see Listing 5.2) in SMT-LIB looks as in Listing 5.16.

```
1 (declare−datatypes ((tTerm 0)) (
2   ((cTrue)
3    (cFalse)
4    (cIfelse (cIfelse_0 tTerm) (cIfelse_1 tTerm) (cIfelse_2 tTerm))
5    (cZero)
6    (cSucc (cSucc_0 tTerm))
7    (cPred (cPred_0 tTerm))
8    (cIszero (cIszero_0 tTerm))) ) )
```

Listing 5.16: Example of translating an SPL closed ADT to SMT-LIB

The "0" in line 1 in Listing 5.16 encodes that the datatype tTerm has no type parameters. Lines 2 to 8 each encode one datatype constructor. Note that we prefix names of datatypes automatically with "t" and constructor names automatically with "c". These prefixes are just for readability, so that we can more easily distinguish types and constructors within SMT-LIB files. In line 4, we automatically generate three selector names for the three constructor arguments of the Ifelse constructor. For this, we simply append consecutive numbers to the constructor name in order to ensure uniqueness of the generated selector names.

5.5.2. Encoding Function Specifications

To define a function in SMT-LIB, one first needs to declare the function's signature via declare−fun. Next, one declares one axiom for each function equation via assert. SMT-LIB provides constructs for **let** and for if-then-else expressions (ite). Therefore, we can encode **let** and **if** constructs from SPL directly. The only thing that we need to encode ourselves is the order of function equations, i.e. we need to add the negative function patterns as premises of function equations (see Section 5.4.2). For this, we modify the existing transformation step that prepares SPL function specifications for encoding to TPTP, simply omitting the translation of **let** and **if** constructs.

For example, we encode function reduce from our running example of typed arithmetic expressions (see Listing 5.3) as in Listing 5.17.

```
1  (declare−fun freduce (tTerm) tOptTerm)
2  (assert (! (forall ((vt2 tTerm) (vt3 tTerm))
3    (= (freduce (cIfelse cTrue vt2 vt3)) (csomeTerm vt2)))
4      :named reduce−0))
5  (assert (! (forall ((vt2 tTerm) (vt3 tTerm))
6    (= (freduce (cIfelse cFalse vt2 vt3)) (csomeTerm vt3)))
7      :named reduce−1))
8  (assert (! (forall ((vt1 tTerm) (vt2 tTerm) (vt3 tTerm))
9    (=> (and (forall ((vt20 tTerm) (vt30 tTerm)) (or (not (= vt1 cTrue))
10           (or (not (= vt2 vt20)) (not (= vt3 vt30)))))
11         (forall ((vt20 tTerm) (vt30 tTerm)) (or (not (= vt1 cFalse))
12           (or (not (= vt2 vt20)) (not (= vt3 vt30))))))
13     (= (freduce (cIfelse vt1 vt2 vt3))
14       (let ((vot1 (freduce vt1)))
15         (ite (fisSomeTerm vot1)
16           (csomeTerm (cIfelse (fgetTerm vot1) vt2 vt3))
17           cnoTerm)))))
18           :named reduce−2))
19 ...
```

Listing 5.17: Example of translating an SPL function definition to SMT-LIB

Note that in Listing 5.17, we applied domain-specific and logical simplification (see Subsection 8.1.3) in order to simplify the premises of negative patterns. Line 1 in Listing 5.17 encodes the function signature of function reduce. Similarly to the prefixes for types and constructors, we prepend "f" to all function names and "v"

to all variable names. Lines 2 to 4 encode the first function equation. Line 4 names the generated axiom. Analogously, lines 5 to 7 encode the second function equation, and lines 8 to 18 the third. There, lines 9-12 encode the negative function patterns for the third equation as premises.

5.5.3. Encoding Inference Rules and Properties

We encode SPL inference rules and properties simply via assertions in SMT-LIB. Like when compiling to TPTP, we generate predicate functions for encoding typing judgments (see Subsection 5.4.3).

For example, we encode the typing rule Tif from our running example of typed arithmetic expressions (see Listing 5.4) in SMT-LIB as shown in Listing 5.18.

```
1 (assert (! (forall ((vt1 tTerm) (vt2 tTerm) (vT tTy) (vt3 tTerm))
2   (=> (and (fptchecksimple vt1 cB) (fptchecksimple vt2 vT) (fptchecksimple vt3 vT))
3     (fptchecksimple (cIfelse vt1 vt2 vt3) vT)))
4       :named Tif))
```

Listing 5.18: Example of translating an SPL typing rule to SMT-LIB

In the Listing above, the function predicate fptchecksimple encodes typing judgments with two arguments (an expression and a type).

5.6. Summary

We presented an instantiation of our VeriTaS verification infrastructure for conducting type soundness proofs: We provided a core input format SPL and another input format that allows for capturing domain-specific information about type system specifications that is relevant to soundness proofs, ScalaSPL. Furthermore, we presented basic tactics for creating proof steps for type soundness proofs. And finally, we connected different verifiers to VeriTaS by implementing encoding strategies from SPL to TPTP and SMTLIB, two main input formats for existing automated provers. We designed all of these components so that they may be reused for other verification domains, with no or only small modifications.

At this point in the thesis, we are able to interactively generate a proof graph for a type soundness proof by manually applying tactics. We are now also able to implement automated proof strategies for type soundness proofs of DSLs, which we demonstrate in the following chapter.

Chapter

6

Automated Generation of High-level Proof Structures

In Chapter 5, we instantiated the VeriTaS verification infrastructure so that we may use it for developing proofs in our target verification domain. In particular, we connected different existing automated verifiers that allow us to verify low-level proof steps encoded within proof graphs.

In this chapter, we automatically generate such low-level proof steps by constructing proof graphs via proof strategies. We focus on our target domain of type soundness proofs of type systems of simple DSLs, but also explain how our strategies can be reused for other verification domains. We use two auxiliary components for abstracting over the domain-specific aspects of specifications: domain-specific knowledge from specification annotations and what we call augmented call graphs. We implement proof strategies on top of these two components.

We first describe the general interplay between the components for the generation of proof graphs and how they are connected to the overall architecture of VeriTaS (Section 6.1). Next, we describe both the domain-specific knowledge annotations (Section 6.2) and augmented call graphs (Section 6.3) in more detail. Finally, we present concrete implementations of proof strategies for our target verification domain of type soundness proofs for DSLs (Section 6.4), using our running example of typed arithmetic expressions.

Remark 6.1. The material in this chapter is unpublished. Some parts (collection of domain-specific knowledge and augmented call graphs) build on material from the master's thesis of Pacak [Pac18], which the author of this dissertation supervised. ◊

6.1. Overview of Generation Approach

Figure 6.1 visualizes the overall architecture of VeriTaS as a whole. The high-lighted components in the middle of the figure ("Augmented Call Graphs", "Domain-specific

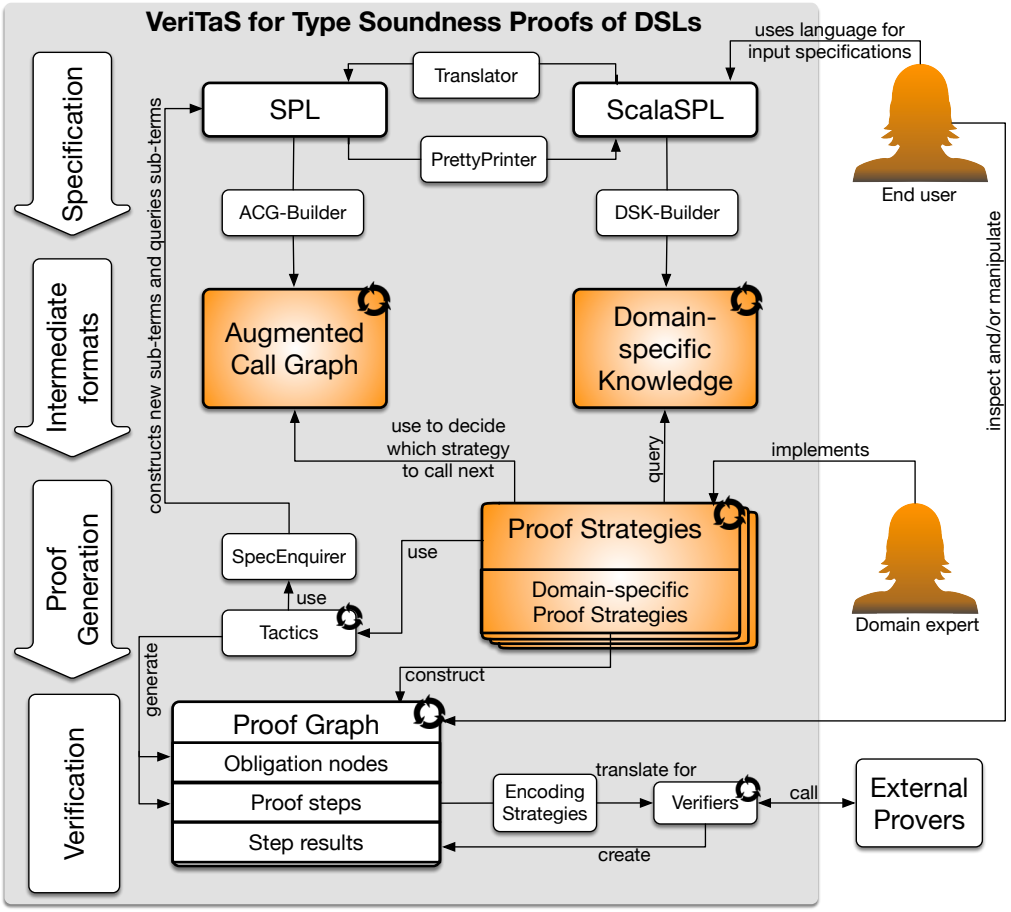


Figure 6.1.: Overview of the overall architecture of VeriTaS, including components for the automated generation of proof graphs

Knowledge", and "Proof Strategies") are the components that we will focus on in this chapter. The remaining parts were introduced in the two previous chapters. In the figure, we deliberately focus on the instantiation of VeriTaS for our target verification domain. When domain experts instantiate VeriTaS for other verification domains, they may employ an equivalent architecture. In Figure 6.1, the small circular symbol with the three bent arrows indicates which components are parametric in a specification format. These components may, to some degree, be reused when instantiating VeriTaS for other verification domains.

In the remainder of this section, we will explain the four vertical parts of Figure 6.1 ("Specification", "Intermediate formats", "Proof Generation", "Verification") from top to bottom, focusing on the interplay and high-level intuition of the different components.

6.1.1. Specification

For the specification of input problems from our target verification domain, end users may employ ScalaSPL (which is a subset of Scala) or also, if preferred, our own internal core specification language SPL. We introduced SPL and ScalaSPL in detail in Section 5.1. SPL is internally used as an intermediate language by various components of the overall architecture. We provide a translation from ScalaSPL to SPL so that all components implemented for SPL may be used for ScalaSPL specifications as well. Additionally, we provide a pretty-printer that prints SPL terms in ScalaSPL so that end users who use ScalaSPL may inspect any terms generated in SPL (such as proof goals) in their original input format.

Both SPL and ScalaSPL are general enough that they may be re-used and adapted for certain other verification domains than our target verification domain, especially for the verification of other properties of programming languages beyond type soundness. For arbitrary verification domains, such as for example the verification of cryptographic protocols, other domain-specific input formats may be more sensible. Domain experts in these domains may add their own input formats and use them with all of or parts of the VeriTaS components that we provide. Domain experts may choose to also use SPL as an intermediate language for their own format in order to be able to reuse as much as possible of the existing infrastructure.

6.1.2. Intermediate formats

We use two components for collecting and organizing proof-relevant information from specifications: augmented call graphs and a collection of domain-specific knowledge, high-lighted in the middle of Figure 6.1. Both components are intermediate formats for the subsequent proof generation.

Our approach here is similar to the approaches used in static program analyses (e.g., data-flow analyses): Here, a program is typically translated into an intermediate format that focuses on the aspects of a program that are relevant for the analyses to be performed. Most approaches first construct a graph that represents the call structure and/or control-flow structure of a program: *Call graphs* [Ryd79] graphically

represent which function in a program calls which other function. *Control-flow graphs* (CFG) [All70] graphically represent the control-flow structure of a function, i.e., where the structure of a function contains a branch or a loop and which values of variables may be available on which particular path. A static program analysis then works purely on the call graph and/or CFG of a program rather than on the program itself. Some particular analyses require further information on the program to be given and/or constructed in order to perform the desired task. For example, an information-flow analysis needs to know for every variable that appears in a program whether it is classified as "secret" (high) or as "public" (low) [Den76; DD77].

Our augmented call graphs use concepts both from control-flow graphs and from call graphs. An ACG-builder generates them automatically from an SPL specification for the subsequent proof generation. Augmented call graphs are structured so that an automated top-level proof strategy may use the structure of such a graph to decide which strategy to call next. We will go into more details regarding the structure and the automated construction of augmented call graphs in Section 6.3. Section 6.4 will go into more details on how automated proof strategies use augmented call graphs.

Our proof strategies also query a collection of domain-specific knowledge, which is automatically constructed by a DSK-builder who collects and transforms domain-specific annotations given in a ScalaSPL specification. Section 6.2 will go into more detail regarding the extensible system for domain-specific annotations in ScalaSPL, which annotations we use for our target verification domain, and how the annotated information is grouped within a collection of domain-specific knowledge.

The main advantages of first translating a problem specification into an intermediate format and of collecting relevant domain-specific knowledge in a separate structure instead of operating on the specifications directly are

1. The implementation of any proof strategies can purely focus on essential concepts. The implementation of the strategies does not need to parse specifications itself or refer to implementation details from the specification format. This reduces the overall complexity of the strategies. Domain experts who implement new proof strategies for existing specification formats can focus on understanding the structure of augmented call graphs and on what the collection of domain-specific knowledge contains.

2. Implemented proof strategies may be reused with other input specifications formats and/or different verification domains, if fitting.

These advantages correspond to the advantages of intermediate formats used in static program analyses: The intermediate formats allow for reducing the complexity of the analyses and ease porting them to different language versions.

Both the component augmented call graphs and the component domain-specific knowledge are parametric in an input specification format. We provide concrete implementations for SPL resp. ScalaSPL, but the largest part of the implementation of both components is independent of a concrete format. Hence, domain experts may reuse both components with their own specification formats by providing

appropriate builders (some generic parts of the ACG-builder may also be reused for this purpose).

6.1.3. Proof Generation

For the automated generation of proof graphs, domain experts implement proof strategies in VeriTaS. Proof strategies can be general ones that construct proof graphs without the aid of any intermediate formats. Domain-specific proof strategies, in our architecture, are proof strategies that make use augmented call graphs and/or query the collection of domain-specific knowledge. Proof strategies may call each other. For example, a top-level proof strategy may traverse a given augmented call graph and decide during this traversal which lower-level proof strategies to call. Ultimately, proof strategies at the lower end apply tactics to construct proof graphs.

We introduced tactics in Section 5.2: Tactics create sub-obligation nodes in a proof graph and proof steps for the subsequent verification of a proof graph. The basic tactics that we implemented when instantiating VeriTaS for our target verification domain are independent of a concrete format for input specifications. For constructing obligations, tactics employ a `SpecEnquirer` which offers methods for querying and constructing sub-terms of SPL. For other specification formats, one may implement a corresponding `SpecEnquirer` and reuse the tactics we provide.

6.1.4. Verification

Proof strategies ultimately construct a proof graph, which is the component that constitutes the conceptual base of VeriTaS. We introduced the concept and implementation of proof graphs in detail in Chapter 4. Proof graphs themselves represent high-level proof structures. Right after their generation, all proof steps within proof graphs are unverified. Hence, a generated proof graph by itself does not yet constitute a proof.

To verify a proof graph, the encoding strategies that we introduced in Chapter 5 translate the problems represented by proof steps from SPL to the input formats of various external ATPs and SMT solvers, which verifiers within VeriTaS call. The return logs from these external provers are processed into step results that are saved within the original proof graph. End users may inspect all parts of a proof graph, notably step results and refine proof graphs.

6.2. Collecting Domain-Specific Knowledge

Domain-specific knowledge in input specifications may be any information that is specific to a certain verification domain and relevant for automatically generating proof graphs of properties. For instance, domain-specific knowledge could be a categorization of functions used in the chosen verification domain, additional information on the nature of certain functions (e.g. whether the function is recursive or may fail), relations between different function arguments, or links between functions and properties.

We start with giving an example for domain-specific information in an example specification from our target verification domain. There, we use domain-specific annotations in ScalaSPL to enable end users to categorize function specifications and link function specifications to auxiliary properties. Afterwards, we introduce the components that we provide in VeriTaS for collecting domain-specific knowledge in detail.

6.2.1. Example: Domain-Specific Annotations for Type System Specifications

We consider the ScalaSPL specification of our running example, a type system for simple arithmetic expressions (see Section 5.1.2). We add the following annotations to the top-level reduction function and to the top-level properties within the specification:

```
1  //reduction semantics for simple Boolean and arithmetic terms
2    @ProgressProperty("Progress")
3    @PreservationProperty("Preservation")
4    @Recursive(0)
5    @Dynamic
6    def reduce(t: Term): OptTerm = t match {
7      case Ifelse(True(), t2, t3) => someTerm(t2)
8      case Ifelse(False(), t2, t3) => someTerm(t3)
9      case Ifelse(t1, t2, t3) =>
10       val ot1 = reduce(t1)
11       if (isSomeTerm(ot1))
12         someTerm(Ifelse(getTerm(ot1), t2, t3))
13       else
14         noTerm()
15         ...
16
17   ...
18
19   @Property
20   def Progress(t: Term, T: Ty): Unit = ...
21
22   @Property
23   def Preservation(t: Term, T: Ty, t2: Term): Unit = ...
```

Listing 6.1: Example: Annotations for the "reduce" function in typed arithmetic expressions

The `@Property` annotations in line 19 and 22 of Listing 6.1 are mandatory to mark that the definitions of `Progress` and `Preservation` are not definitions of functions (as the Scala syntax would indicate) but properties. The remaining annotations are not mandatory, but necessary for the correct functioning of the automated proof strategies that we are going to introduce in Section 6.4.

The annotations in line 2 and 3 link the property named `Progress` as a progress property for function `reduce` resp. the property named `Preservation` as a preservation property for function `reduce`. The automated strategies use this information

for inserting the correct properties in the correct place when constructing a proof graph. The annotation in line 4 indicates that `reduce` is a *recursive* function, with its first argument (argument at position 0) being the decreasing argument. The automated proof strategies use this information to decide where to apply induction and for which variable. The annotation in line 5 indicates that `reduce` is a function from the *dynamic* semantics of the language specification. We would mark a function that is part of the *static* semantics of a language specification with `@Static`. The automated strategies will use this information to decide for which functions an augmented call graph is generated and used for proof generation.

We annotate the remaining auxiliary functions from the specification of our running example in the same way. Notably, we use the annotations `@ProgressProperty` and `@PreservationProperty` to link auxiliary functions to properties about this function. The fully annotated ScalaSPL specification of our running example can be found in Appendix A.1.

6.2.2. Collection Infrastructure

For the general case, there are two ways of how domain-specific knowledge may be obtained from input specifications: The knowledge may be automatically extracted from "raw" input specifications, or domain experts may require end users to annotate input specification with the desired information. The latter approach requires more specification effort from end users, but is potentially more flexible: Automatically extracting domain-specific knowledge from input specifications may not work as desired for arbitrary user specifications within a certain verification domain. By requiring an end user to provide the desired information via annotations, domain experts can easily cover a large number of individual specification patterns and focus their attention on implementing more powerful domain-specific proof strategies.

A collection of domain-specific knowledge in VeriTaS groups the domain-specific knowledge from a specification into a trait with fields in which one may look up the annotated information for a specific function definition. A DSK-builder traverses a specification and constructs a collection of domain-specific knowledge accordingly, potentially transforming the knowledge as required by the collection.

Implementation For our target verification domain, we use the following generic trait to collect domain-specific knowledge:

```
1 trait DomainSpecificKnowledge[Type, FDef, Prop] {
2   def recursiveFunctions: Map[FDef, (Type, Seq[Int])]
3   def progressProperties: Map[FDef, Set[Prop]]
4   def preservationProperties: Map[FDef, Set[Prop]]
5   def auxiliaryProperties: Map[FDef, Set[Prop]]
6
7   def properties: Set[Prop]
8   def staticFunctions: Set[FDef]
9   def dynamicFunctions: Set[FDef]
10
11  ...
```

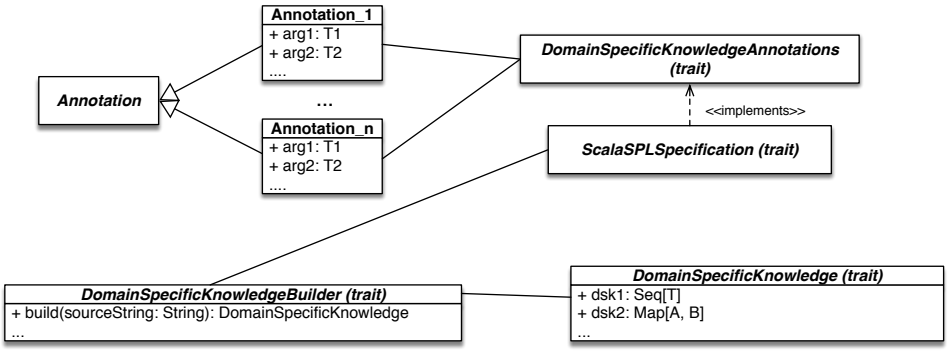


Figure 6.2.: UML schema for abstract components of ScalaSPL's extensible annotation system

```
12 }
```

Listing 6.2: Trait for `DomainSpecificKnowledge` for type soundness proofs

This trait can be reused for collecting the domain-specific knowledge of type system specifications in other formats than SPL/ ScalaSPL. We instantiate its type parameters (`Type`, `FDef`, and `Prop`) with the appropriate case classes from SPL for our purposes. Our DSK-builder collects the domain-specific annotations in ScalaSPL specification and constructs the attributes of trait `DomainSpecificKnowledge` in Listing 6.2.

For instance, for our running example attribute `recursiveFunctions` will contain an entry that maps the SPL function definition of function "reduce" to a pair consisting of the SPL type of "reduce" and the sequence of integers that mark the position of the decreasing argument of the recursive function (i.e. the sequence of integers provided as arguments to the `@Recursive` annotation). Attribute `progressProperties` will contain an entry that maps the SPL function definition of function "reduce" to a set that contains the SPL definition of the property "Progress".

For convenience, trait `DomainSpecificKnowledge` from Listing 6.2 also features functions for accessing entries within the maps via function name only.

6.2.3. ScalaSPL's Extensible Annotation System

As we have already seen in the example above, ScalaSPL features a system for user annotations. This annotation system is *extensible*, i.e., domain experts may flexibly add their own domain-specific annotations to ScalaSPL. The extensible annotation system for ScalaSPL was developed by Pacak in his master's thesis [Pac18].

Implementation Figure 6.2 summarizes the main abstract components of ScalaSPL's extensible annotation system: Concrete annotations extend Scala's abstract class `Annotation`. The annotations may have multiple arguments of different types to encode additional information. We group all annotations that are relevant for a particu-

lar verification domain within a Scala trait `DomainSpecificKnowledgeAnnotations`, which contains additional helper methods, e.g. for looking up certain information within annotations. A concrete ScalaSPL specification has to extend the trait `ScalaSPLSpecification`. If a ScalaSPL specification wants to make use of the defined concrete annotations `Annotation_1` to `Annotation_n` then one has to mix in the trait `DomainSpecificKnowledgeAnnotations` in addition. The DSK-builder (trait `DomainSpecificKnowledgeBuilder`) takes a concrete ScalaSPL specification that extends the desired `DomainSpecificKnowledgeAnnotations` as a source string. The build method processes this source string via reflection and collects the specification elements that are annotated with certain annotations. Ultimately, the build method groups the elements into the attributes defined within the `DomainSpecificKnowledge` trait for the verification domain in question.

Domain experts can easily introduce their own ScalaSPL annotations by extending the abstract class `Annotation` and by implementing their own variants of the traits in Figure 6.2 resp. by extending existing traits. To process their new annotation appropriately, they need to introduce a corresponding attribute in their `DomainSpecificKnowledge` trait and to extend the `build` method within their `DomainSpecificKnowledgeBuilder` trait accordingly.

Within a ScalaSPL specification, we can use annotations as follows:

```
1 @Annotation_1(arg1, arg2,....)
2 def f(arg: T1): T2 = ...
```

Listing 6.3: Usage of annotations in ScalaSPL (abstract)

More details on ScalaSPL's extensible annotation system and on how to add a custom annotation can be found in Pacak's master's thesis [Pac18].

6.2.4. Annotations for Type Soundness Proofs

Finally, we list the annotations that we provide for implementing proof strategies for our target verification domain. The annotations are based on the annotations developed in Pacak's master's thesis [Pac18]

We provide the following mandatory, general annotations (the information given by these annotations is important for the translation from ScalaSPL to SPL):

- **@Partial** marks whether a function definition is partial, i.e. does not provide a case definition for each possible argument. This annotation is mandatory, since its information is important for the translation from ScalaSPL to SPL.
- **@Axiom** marks whether a definition (marked `def` within ScalaSPL) is an axiom. For example, each typing rule should be marked as @Axiom. This annotation is mandatory, since its information is important for the translation from ScalaSPL to SPL.
- **@Property** marks whether a definition (marked `def` within ScalaSPL) is a property. This information is important for the translation from ScalaSPL to SPL.

Furthermore, we provide the following optional, domain-specific annotations:

- **@Recursive(pos*)** marks whether a function is recursive or not. The `pos` argument can be one or more integers that indicate the positions of top-level arguments that *decrease* for each recursive call, starting with 0 for a function's first argument. E.g. the annotation "@Recursive(0, 1)" indicates that the annotated function decreases in its first *and* in its second argument. We do not support nesting of position descriptions, i.e. if the decreasing argument appears as a constructor argument that is in turn an argument of the top-level function. We use this annotation within the automated strategies to decide whether we have to apply a structural induction tactic, and on which function variables we need to apply induction (details follow in Section 6.4).

- **@Dynamic** marks whether a function belongs to the (dynamic) reduction semantics of a language specification. That is, the top-level reduction function (typically named "reduce") should be marked @Dynamic, as well as every function that `reduce` calls. The automated proof strategies use this information for example for retrieving function definitions for which augmented call graphs will need to be constructed (details follow in Section 6.4).

- **@Static** marks whether a function belongs to the (static) type system of a language specification. Every auxiliary function called from the typing rules should be marked @Static. This information may be used by a future lemma generation approach (more in Chapter 10).

- **@ProgressProperty(property_name)** is an annotation for a function definition and links a property to a function as its "progress property". The annotation's argument is the name of the property in the ScalaSPL specification. A "progress property" states under which premises (typically, "static" conditions, i.e. conditions for functions marked @Static) the function in question will definitely return a result. The automated proof strategies use this information when generating lemma applications within a proof graph for looking up appropriate lemmas (details follow in Section 6.4).

- **@PreservationProperty(property_name)** is an annotation for a function definition and links a property to a function as its "preservation property". The annotation's argument is the name of the property in the ScalaSPL specification. A "preservation property" typically states which "static" conditions (i.e. conditions for functions marked @Static) the function in question will preserve for the result it returns. The automated proof strategies use this information when generating lemma applications within a proof graph for looking up appropriate lemmas (details follow in Section 6.4).

- **@AuxiliaryProperty(property_name)** is an annotation for a function definition with which one may link an arbitrary property to a function that is important for a proof, but can neither be categorized as "progress" nor as "preservation" property. The automated strategies will also take these

properties into account when generating lemma applications within a proof graph (details follow in Section 6.4).

Some of these annotations may also be useful in other verification domains and may easily be re-used.

Note The proof strategies as well as the different translations (ScalaSPL to SPL, SPL to TPTP/SMTLIB) rely on completely and correctly annotated ScalaSPL specifications by the end user. We do not check at any point whether the provided annotations "make sense". Adding such checks would be conceptually straightforward, but would require some further engineering effort that was not in the scope of this thesis. If annotations are missing or attributed incorrectly to definitions, the strategies as well as the translations will nevertheless attempt to generate results, but they are likely to generate incorrect or at least incomplete results.

6.3. Augmented Call Graphs and Their Construction

We use augmented call graphs (ACGs) as an intermediate format to structure the individual parts of a function specification into a format suitable for proof generation. As such, the format deliberately abstracts over information in the definition of a function that is not relevant for proof generation. Augmented call graphs have nodes for representing structural case distinctions, boolean case distinctions, and function calls that occur within the definition of a function. For each function definition within a problem specification, we may generate one (unique) augmented call graph. We impose certain requirements on the overall structure of an augmented call graph, which allows us to implement proof strategies that follow a single, universal schema. We will go into more details regarding the structure of augmented call graphs later in this section.

Augmented call graphs contain concepts from call graphs as well as from control-flow graphs. They contain the information which function calls which other function, like call graphs. They also encode the general control flow of a function, like control-flow graphs. However, they are not the direct combination of call graphs and control-flow graphs, such as *interprocedural control-flow graphs* [LR91; LR92] are: Each augmented call graph represents the structure of a single function only, and the augmented call graphs of different functions are not directly linked to each other. The name "augmented" call graph in VeriTaS is "historically grown". Early conceptual visions operated with a standard call graph, and then we added structural information about a function definition. The additional information allowed for constructing more fine-grained proof steps.

The augmented call graphs used in this thesis build directly on the data structure with the same name introduced by Pacak in his master's thesis [Pac18], with some refinements and improvements of the generation algorithm that we will name below.

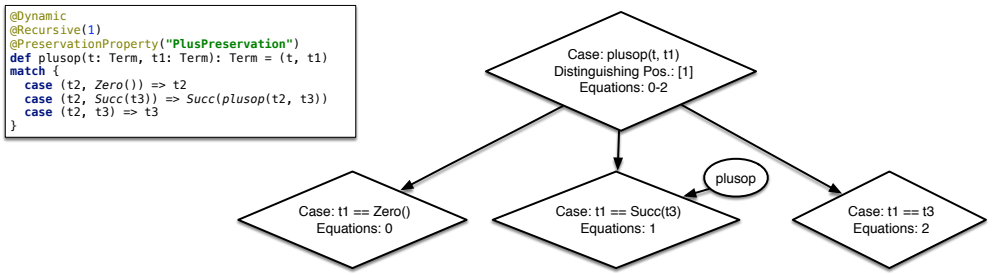


Figure 6.3.: Example of a complete augmented call graph: a simple plus operation

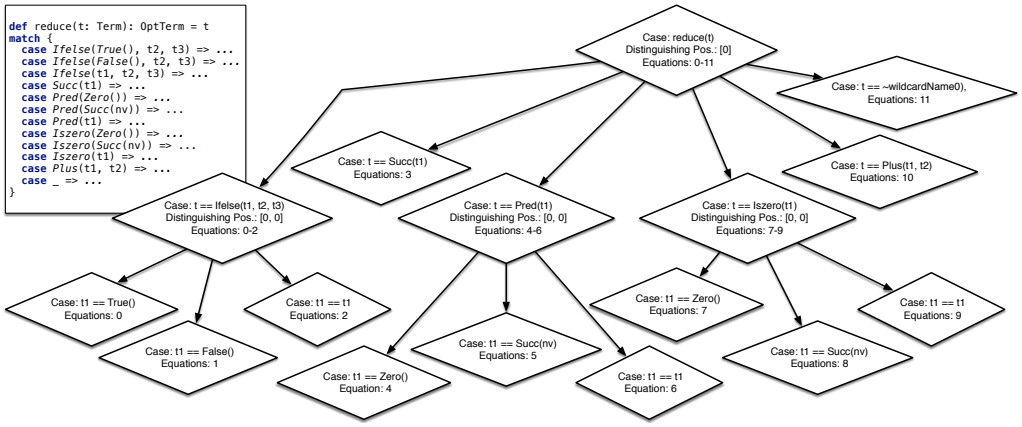


Figure 6.4.: Excerpt of augmented call graph of reduce function for typed arithmetic expressions: structural distinctions

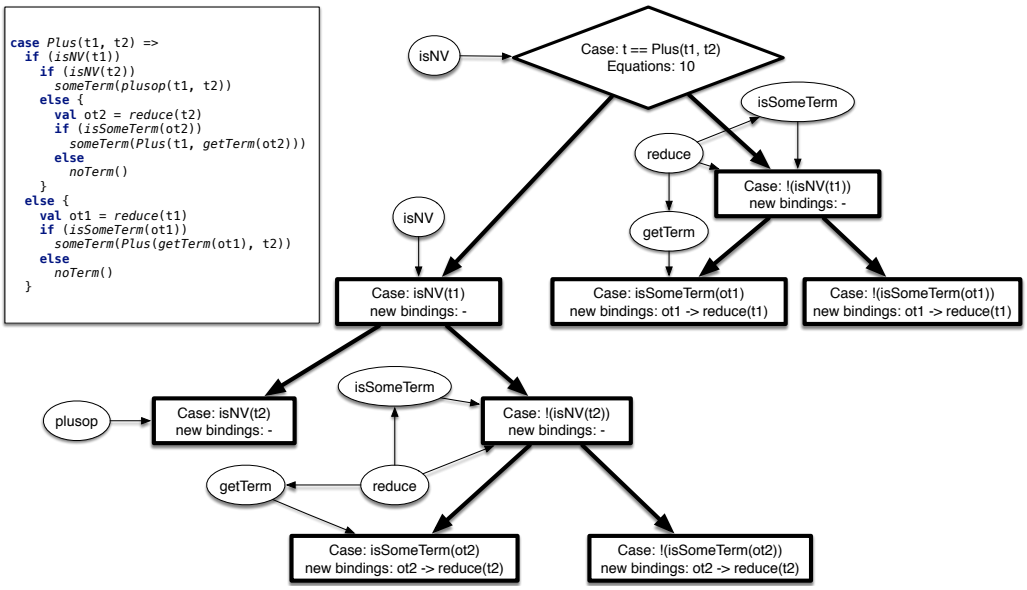


Figure 6.5.: Excerpt of augmented call graph of extended reduce function for typed arithmetic expressions: plus case

6.3.1. Examples of Augmented Call Graphs

Like in the previous section, we start with concrete examples, using our running example of typed arithmetic expressions. We show and explain how the generated ACGs for selected functions from our example specification look like. We visualize each example in a figure that contains, on the left-hand side, the relevant excerpt of the function definition (in ScalaSPL) for which we visualize the ACG. Within the ACGs, we visualize structural distinction nodes as diamonds, boolean distinction nodes as boxes, and function call nodes as ellipses.

To show a first small, but complete example of an ACG, we extend our running example with a small recursive plus operation on natural numbers. Figure 6.3 shows on the left-hand side the fully annotated ScalaSPL specification of this operation: We will use `plusop` within our reduction semantics of arithmetic expressions extended with a syntactic `Plus` constructor. Hence we annotate `plusop` as `@Dynamic`. Furthermore, its specification is recursive in its second argument, hence the annotation `@Recursive(1)`. When we later generate a progress and a preservation proof for the type system of our extended running example, we will need a "preservation" property for `plusop` as auxiliary property. We will show this property in the examples below. It is called "PlusPreservation", hence we add the annotation `@PreservationProperty(''PlusPreservation'')` to link this property to `plusop`.

The right-hand side of Figure 6.3 shows the complete, automatically generated ACG for `plusop`. Its root node is a structural distinction node that contains all three function equations (equations with index 0 to 2) of `plusop` and a generic function call to `plusop` as common expression ("Case: ..."). The ACG-builder detects that the next position ("Distinguishing Pos.:...") in which we have to distinguish the three equations it the second argument position, i.e. here argument `t1`, the argument at index 1 (argument indices start with 0). Distinguishing the three equations further yields three structural nodes, each of which contains only one equation (given by its index). Each of these structural distinction nodes contains an equation for `t1`, correctly assigning the respective expressions from each function equation.

The first and the third case within `plusop` do not contain any more relevant expressions for the ACG generation. The second case contains the recursive function call to `plusop`, hence we add a function call node as parent to the structural distinction node for that case. The direction of the edge between the function call node for `plusop` and the structural distinction node for the second case models that the result of that function call is used within the case.

We decided to keep ACGs acyclic, hence we generate a separate function call node for the recursive `plusop` call instead of creating an edge to the ACGs root node. This will simplify the implementation of proof strategies that use ACGs: The strategies will not have to detect and correctly treat cycles in ACGs. Another option would have been to omit function call nodes for recursive calls. We decided to keep them since they may give hints to proof strategies as to where exactly an induction hypothesis probably needs to be applied. Whether this information can be sensibly passed on to external provers depends on the provers used. We discuss

this further in Section 6.4.

Function `plusop` only has a small number of function equations with no overlapping patterns. The `reduce` function of our running example (extended with a `Plus` constructor) is larger and more complicated.

Figure 6.4 visualizes the sub-tree of structural distinction nodes of the ACG for `reduce`. The left-hand side of Figure 6.4 shows a fragment of the ScalaSPL specification of `reduce`, focusing on the patterns of the function equations. The corresponding tree of structural distinction nodes start withs a root structural distinction node that contains all 12 function equations. Its 6 direct children group the cases for each of the 5 term constructors (including the new `Plus` constructor) and for the default case.

For the constructors `Succ` and `Plus` as well as for the default case, there is only equation, hence no further structural distinction is necessary. The remaining cases group several more sub-cases. For instance, the left-most case for `Ifelse` contains the first three function equations of `reduce`. The next distinguishing position, `[0, 0]`, marks the first argument within the first argument of a top-level `reduce` call. In this example case, this refers to the `t1` argument within an `Ifelse` term, i.e. the guard. There, we distinguish 3 further cases (`True()`, `False()`, and another term unequal to `True()` or `False()`). This hierarchical structure of structural distinction nodes at the top of each ACG helps to compute the correct necessary case distinctions within proofs.

Finally, we illustrate the binary sub-trees that we generate for each individual function equation in an ACG and the additional function call nodes that we add to the basic structure of each ACG. For this, we study the new case for the `Plus` constructor that we added to the `reduce` function of our running example. Figure 6.5 shows the full ScalaSPL specification of this case in the box at the left-hand side. The excerpt of the ACG for the `reduce` function on the right-hand side highlights the binary sub-tree for the structural parts of the `Plus` case in bold: The root node of the highlighted sub-tree is the structural distinction leaf for the `Plus` case from Figure 6.4. The direct children of the root node distinguish the cases `isNV(t1)` and `!isNV(t1)`. Their children again distinguish Boolean cases as in the code.

Each boolean distinction also contains the information whether there are new variable bindings that are active within this case. For instance, the rightmost boolean distinction node in Figure 6.5 (case `!isSomeTerm(ot1)`) contains the new binding for `ot1` from the code.

The nodes within a binary sub-tree for a case have function call nodes as additional parents. These nodes are added to the ACG if there are function calls within sub-cases. For instance, the root node in Figure 6.5 has a function call to `isNV` as parent, since this function is called to decide the top-level boolean distinction, i.e. the corresponding call refers to the `isNV(t1)` call in the ScalaSPL specification. Case `isNV(t1)` then has *another* function call to `isNV` as parent, this time representing the call `isNV(t2)` that occurs top-level within this case.

Case `!isNV(t2)` illustrates the flow of results between different function calls, also represented within ACGs: At the top-level, this case calls functions `isSomeTerm` and `reduce`. The call to `isSomeTerm` uses the result of the call to `reduce`, hence the edge

from the function call node for `reduce` to the function call node for `isSomeTerm`. Finally, the leaf case `isSomeTerm(ot2)` has a function call node for `getTerm` as parent, which uses the result of the previous function call for `reduce` - hence the edge from the previous function call node for `reduce` to the function call node for `getTerm`. Note that in such a case as this one, it is important to not duplicate the function call node for `reduce`, since the ACG shall preserve the information that both `isSomeTerm` and `getTerm` use the result from the *same* function call.

6.3.2. Definition and Structure

We will now give an exact definition of ACGs and their structure. We designed ACGs for our target verification domain, but their structure is universal enough to be useful for other verification domains as well: In many domains, it is important for automated proof construction to know where which kind of case distinction in a definition occurs so that proofs of properties over the corresponding functions can be constructed with appropriate distinction steps. Recursive function calls indicate where it may be necessary to apply an induction hypothesis, while calls to other functions indicate where it may be necessary to apply an auxiliary lemma.

Augmented call graphs consist of three kinds of different nodes: structural distinction nodes, boolean distinction nodes, and function call nodes.

Structural distinction nodes Nodes for structural distinctions group function equations within function definitions in different structural cases.

Definition 6.1 (Structural distinction node)**.** A *structural distinction node* contains an indexed set of function equations for a specific case from the definition of the original function. Furthermore, it contains an expression that is common to all cases within the node as well as a sequence of integers that denotes the argument position (from the top-level function call) where the present case may be refined further. ◊

A structural distinction node that is the root of a structural distinction sub-tree will always contain all function equations from the function definition for which the present ACG is created. The common argument expression is a generic call to this function with fresh argument variables. Structural distinction children contain less function equations and refined common argument expressions, down to structural leaves that contain only one function equation.

Compared to the definition of structural distinction nodes within Pacaks master's thesis [Pac18], we refined common argument expressions and added argument positions for further refinements of the structural distinction.

Boolean distinction nodes Nodes for boolean distinctions model distinctions of exactly two cases according to a predicate that is true or false.

Definition 6.2 (Boolean distinction node)**.** A *boolean distinction node* contains the boolean expression on which the distinction is made as well as the resulting expression for this case from the function definition. Additionally, it contains new variable bindings which apply for the distinguishing boolean expression as well as

for the resulting expression. To ensure uniqueness of Boolean distinction nodes, they also contain the index of the function expression where the boolean distinction occurs as well as an integer that represents the nesting level of the current case (will be greater than 0 if nested boolean distinctions occur within a case). ◊

Compared to the definition of structural distinction nodes within Pacaks master's thesis [Pac18], we added variable bindings, the function equation index, and inner nesting levels.

Function call nodes Nodes for function calls abstract from function calls that occur within a particular case.

Definition 6.3 (Function call node)**.** A *function call node* contains the name of the function that is called. To ensure uniqueness of function call nodes, they also contain the index of the function equation in which the call occurs as well as an integer that represents the nesting level of the current case (will be greater than 0 if nested boolean distinctions occur within a case). ◊

That is, function call nodes abstract over the concrete argument expressions used for a function call within the definition of a function. Compared to the definition of structural distinction nodes within Pacaks master's thesis [Pac18], we added the function equation index and the inner nesting level.

Now we can define augmented call graphs exactly:

Definition 6.4 (Augmented call graphs (ACGs))**.** An *augmented call graph* is a directed acyclic graph (DAG) that has as nodes structural distinction nodes, boolean distinction nodes, and function call nodes and satisfies all of the following structural criteria:

1. The structural distinction nodes of the ACG form a tree.
2. Structural distinction nodes do not have boolean distinction nodes as parents.
3. All structural distinction nodes that contain more than one function equation only have other structural distinction nodes as parents or as children.
4. All structural distinction nodes that contain only one function equation only have boolean distinction nodes as children.
5. Boolean distinction nodes only have other boolean distinction nodes as children.
6. All function call nodes have at least one child node. Children of function call nodes are either boolean distinction nodes or structural distinction nodes with exactly one function equation or other function call nodes.

◊

The criteria from above ensure that ACGs have the following unified structure: The "upper part" of every ACG is a tree made of structural distinction nodes. When ignoring function call nodes, the leaves of this structural distinction tree

form consist of binary subtrees which always have as root a structural distinction node with exactly one function equation and as remaining nodes boolean distinction nodes. The nodes within these binary subtrees may have function call nodes as parents, which may themselves have function call nodes as parents. The direction of the edges from function call nodes to other function call nodes resp. to structural distinction leaves and boolean distinction nodes models that results of these function calls are used by the child node.

Having this unified structure is beneficial for implementing automated proof strategies using ACGs: Strategies only need to treat a clearly defined and limited number of cases when constructing proof graphs.

6.3.3. Automated Construction of Augmented Call Graphs

We sketch the main steps for automatically constructing ACGs from SPL specifications (or of course also from ScalaSPL specifications translated into SPL). In our instantiation of VeriTaS, we implemented class `AugmentedCallGraphBuilder` for general steps that are independent from the input format and instantiated the abstract steps in class `VeritasAugmentedCallGraphBuilder` for SPL.

Step 1: Hierarchical grouping of function equations by their pattern The construction algorithm for an ACG for a given function definition first constructs a root structural distinction node that meets the criteria for ACG root nodes defined above. For this, the algorithm constructs a generic function call with fresh variables as common argument expression. As distinction position, it either takes the position argument from a `@Recursive` annotation that the function may have, or uses the first possible argument position as default.

Next, the algorithm recursively refines the structural distinction nodes: Starting from the root node, it first checks whether the current structural distinction node contains more than one function equation. If yes, we group the given function equations according to their argument patterns and the given indication for distinguishing argument positions. For each group we obtain, we compute a common argument expression, i.e. an equation with fresh variables that refines one or more variables from the argument expressions of parent structural distinction nodes. Also, we refine the argument position within the current node to obtain the next position at which the new group will be refined further (may be `None` if the new group only contains one function equation).

Finally, we create new structural distinction nodes with the newly computed information and add them to the ACG. If a new group still contains more than one equation, we recursively call the function that computes step 1 on the newly added structural distinction node.

The author of this thesis completely reimplemented step 1 with regard to what Pacak describes in his master's thesis [Pac18] to support more complex patterns in function equation and handle argument variables correctly during graph construction.

Step 2: Add Boolean distinction nodes and function call nodes At the end of step 1, we finished constructing the top structural distinction tree of an ACG. The construction algorithm then continues on each structural distinction leaf, inspecting the right-hand side of the corresponding function equation from the original function definition. The algorithm first looks for function calls occurring within variable bindings and guard expressions of boolean distinctions (if expressions) on the top level of the current expression. It also considers inner function calls that occur within argument positions of outer function calls. We add a function call node as parent of the current node for each function call that we find. The algorithm creates edges between function call nodes according to the usage of other functions discovered in argument positions. It constructs a map of variable bindings that is used as auxiliary information for correctly computing all of these edges.

When the algorithm encounters an if expression, it retrieves the two expressions from the two branches and adds a corresponding boolean distinction node to the ACG for each branch. It passes the new top-level bindings found in the current node as arguments to these two new nodes. Finally, the algorithm recursively calls step 2 for each of the new boolean distinction nodes. The next top-level expression considered then becomes the expression within each of the branches.

For step 2, the author of this thesis only introduced minimal changes with regard to the construction algorithm from Pacak's master's thesis [Pac18].

6.4. Designing and Implementing Proof Strategies

Finally, we describe how domain experts may design proof strategies that automatically construct proof graphs, using the information within a collection of domain-specific knowledge as well as from augmented call graphs. We start by describing how the proof strategies operate for our target domain, using our running example of typed arithmetic expressions. Afterwards, we describe the general pattern of the proof strategies we designed and implemented for generating progress and preservation proofs. Finally, we discuss how one may implement proof strategies for other verification domains.

6.4.1. Proof Strategies for Type Soundness Proofs by Example

We consider the automated generation of a progress and a preservation proof for our type system for arithmetic expressions (see Appendix A.1 for the full ScalaSPL specification). An initialization strategy starts with generating two independent root obligations: one with the top-level progress property, and one with the top-level preservation property. Next, we apply a domain-specific proof strategy that implements a top-level generation loop - let us call this strategy *top-level loop strategy* for further reference. The top-level loop strategy receives a function from the dynamic semantics of our specification. We start with the top-level function, `reduce`. The top-level loop strategy first triggers the generation of the ACG for `reduce`.

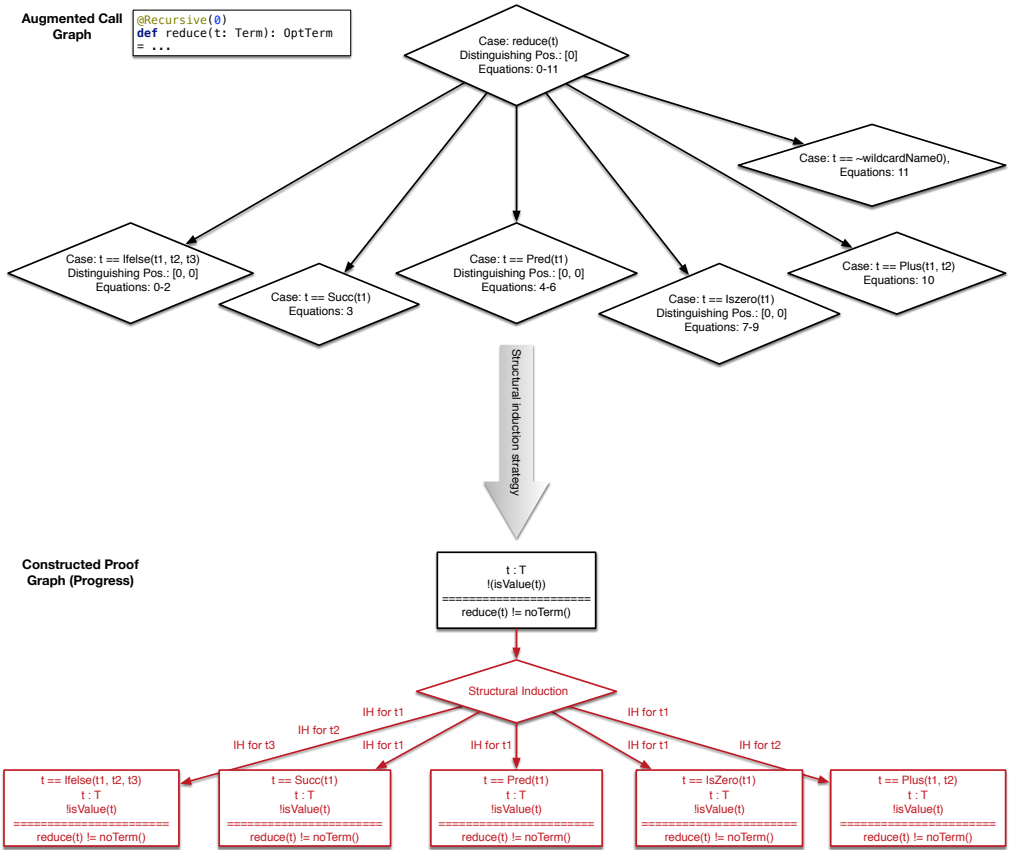


Figure 6.6.: Example of applying strategy for structural induction (excerpt of progress proof of typed arithmetic expressions)

In the upper part of Figure 6.6, we see an excerpt of the ACG generated for `reduce`, namely the first two levels of its sub-tree of structural distinction nodes. The top-level loop strategy inspects the root structural distinction node and notes that it is this node contains more than one equation. Ergo, we need to add either a case distinction or an induction step to the proof graph we are constructing. To decide which of these two steps we actually need, the top-level loop strategy queries the collection of domain-specific knowledge for `reduce` and finds that `reduce` is marked as a recursive function, with its single argument being the argument that decreases with each recursive call. Hence, we apply a general structural induction strategy.

Induction strategy The lower part of Figure 6.6 shows how the structural induction strategy generates induction cases for the progress part of the proof. On top of the depicted proof graph excerpt, we see the top-level progress obligation. We highlight in red the proof step (structural induction) and the new sub-obligations that the strategy generates and adds to the proof graph that is being constructed. The structural induction strategy generates these components by applying the basic structural induction tactic, passing the correct induction variable (here, `t`). We can see in the figure that the individual induction cases are simply generated by adding a case premise to the original obligation. The structural induction tactic makes sure that any necessary induction hypotheses are generated and passed, together with any necessary fixed variables, along the generated proof edges.

The top-level loop strategy also applies the structural induction strategy on the top-level preservation obligation. The resulting structure of the proof graph on the preservation side is analogous to the excerpt at the bottom of Figure 6.6, just with the corresponding preservation obligations.

Note that in our example, every generated induction case corresponds to one structural distinction child in the ACG in the upper part of Figure 6.6. However, there cannot be an induction case that corresponds to the right-most child of the root structural distinction, the "wildcard" case: Structural induction on a particular variable of a structured type is by definition complete.

General case distinctions To continue, the top-level loop strategy matches the newly generated induction cases to the direct structural distinction children of the structural distinction root. The strategy constructs a matching by comparing the case premises of the induction cases to the case expressions in the structural distinction nodes in question. For each induction case, we inspect the corresponding structural distinction node and its children to decide on the next step to construct in the proof graph.

For example, let us consider the `Ifelse` case of the progress proof. The upper part of Figure 6.7 shows the corresponding structural distinction node from the ACG for `reduce` and its three direct children. The lower part of the figure shows, in black, the sub-obligation within the constructed proof graph for the induction case for `Ifelse`. Since we are considering an internal case within a function, we know that we cannot sensibly apply any induction strategy. We note that the structural distinction node

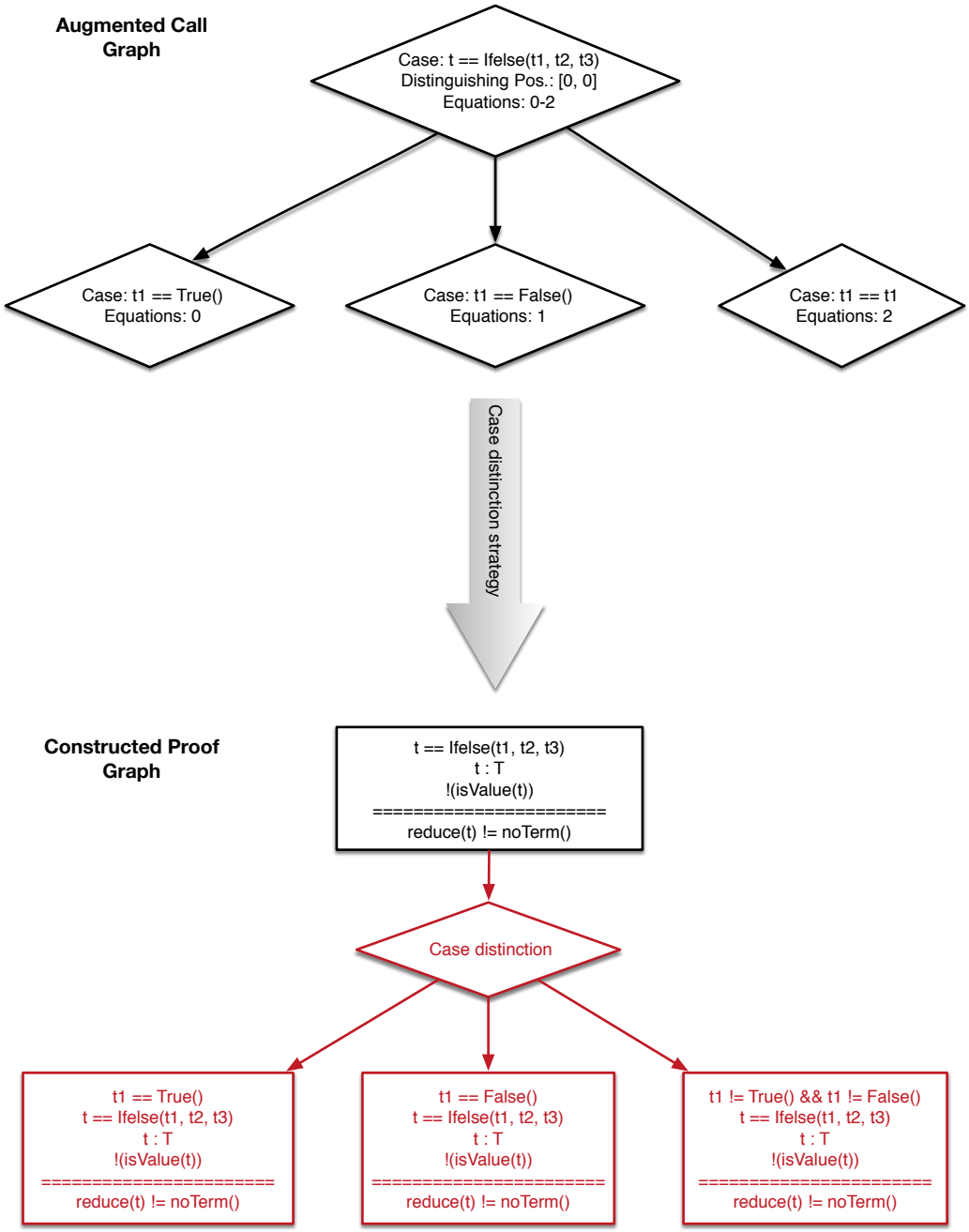


Figure 6.7.: Example of applying a general case distinction for further distinguishing the `Ifelse` induction case within the progress proof for typed arithmetic expressions

for `Ifelse` that we are considering here as three structural distinction children (i.e., sub-cases). So we apply a general case distinction strategy.

The general case distinction strategy calls the case distinction tactic, passing the corresponding case premises. In the lower part of Figure 6.7, we highlight in red the sub-obligations and the proof step that the tactic adds in this case. We see that the tactic generates three sub-obligations, each corresponding exactly to one of the structural distinction children for the ACG above. Each sub-obligation receives the corresponding case equation as an additional premise. The general case distinction strategy makes sure that the order of the cases is correctly encoded within the premises: It adds negative patterns as additional premises for previous cases. We see these additional negative premises in the right-most generated obligation at the bottom of Figure 6.7.

We treat the top-level induction cases for the `Pred` and for the `Iszero` constructor in the same way, since they both also have several structural distinction children. We treat the three cases we discussed in this paragraph analoguouly on the preservation side of the proof graph that is being constructed.

Boolean case distinctions Now we consider the remaining top-level induction cases of the progress proof, the `Plus` case and the `Succ` case. We focus on the `Plus` case - for the `Succ` case, the top-level loop strategy applies analogous steps, both in the progress and in the preservation part of the generated proof graph.

The upper part of Figure 6.8 shows the excerpt of the ACG of `reduce` that corresponds to the `Plus` case. This excerpt is an excerpt from the full ACG for the `Plus` case in `reduce` that we already showed in Figure 6.3. Again, we highlight the nodes corresponding to the control-flow structure of the case in bold (the structural distinction node at the top and the boolean distinction nodes. We ignore the function call nodes for now and focus on this part.

The top-level loop strategy inspects the structural distinction node for `Plus` and sees that it has two boolean distinction nodes as children (in the figure, we omit the then-case, `isNV(t1)`, for simplicity. Hence, we call a general boolean distinction strategy. This strategy adds two new sub-obligations and a new proof step to the obligation in the proof graph for the `Plus` case, via the tactic for boolean distinctions. In the lower part of Figure 6.8, the black part at the top of the excerpt of the visualized proof graph corresponds to this step (the generated sub-obligation for the then-case is again omitted).

Next, the top-level loop strategy inspects the children of the boolean distinction nodes of the ACG. We see that the case for `!(isNV(t1))` has again two boolean distinction nodes as children. So we apply again a the boolean case distinction strategy, which adds the proof step and sub-obligations to the proof graph as highlighted in red in the lower part of Figure 6.8.

Once the top-level loop strategy arrives at a structural distinction node or at a boolean distinction node that does not have any children, it stops. This is for example the case for the `!(isSomeTerm(ot1))` case. The top-level loop strategy simply applies the default Solve tactic in this case.

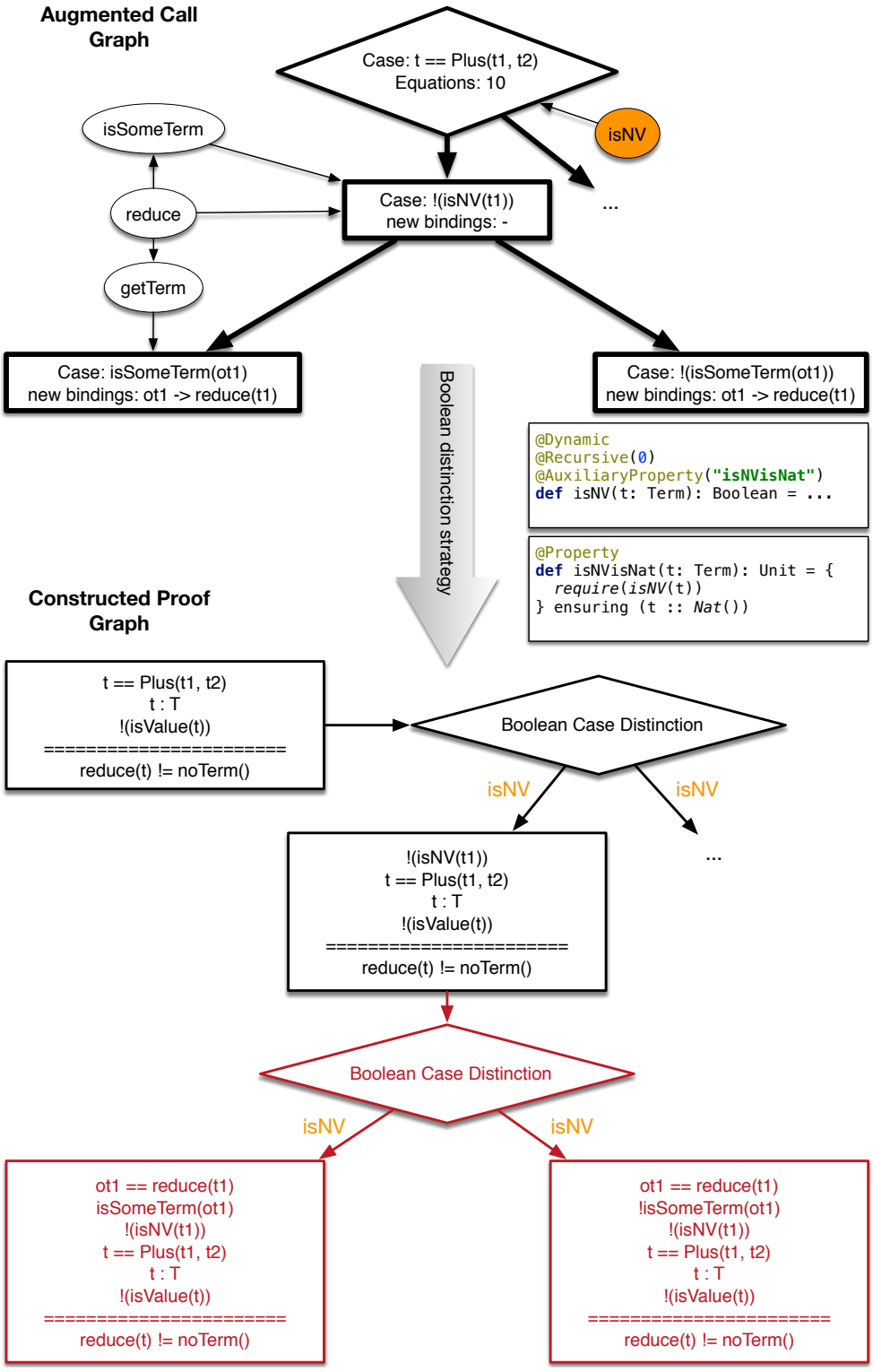


Figure 6.8.: Example of applying a boolean case distinction for further distinguishing cases within the `Plus` induction case and of propagating pending lemma applications in the progress proof for typed arithmetic expressions

Lemma application Up to now, our generated proof graph is a forest that consists of two trees that are not connected to each other and structurally equal - one for the progress proof, one for the preservation proof. Now we will explain how the top-level loop strategy automatically adds lemma application steps. In short, the top-level loop strategy considers the function call nodes within the ACG and adds corresponding lemma application steps. In principle, we can now have the situation that we need a particular auxiliary lemma in several cases, and maybe also both in the progress and in the preservation part of the proof graph. That is, during the addition of lemma application steps, the two trees in our generated proof graph may actually become connected, turning the forest into a directed acyclic graph.

We stay with the `Plus` case and consider again Figure 6.8, where some function call nodes exist in the depicted excerpt of the ACG of `reduce` for the `Plus` case that we did not discuss yet. The structural distinction node for the `Plus` case at the top of the figure has a function call node as parent representing a function call to `isNV` (highlighted in orange in the figure). This function call node appears because of the call to `isNV` in the guard of the if-expression that appears a the top-level within this case: We ask first whether term `t1` is a numeric value.

In the middle of the figure, right of the grey arrow, we see two boxes with excerpts from the ScalaSPL specification of our running example. In the first box, we see the domain-specific annotations for `isNV`: It is marked as a dynamic function, that is recursive in its single argument. Furthermore, the annotation `@AuxiliaryProperty` marks that there is a property named `isNVisNat` about the function `isNV`. We see the full ScalaSPL specification of this property in the box just below. The property states that every numerical value has type `Nat`.

Due to the `@AuxiliaryProperty` annotation, we know that we might have to use property `isNVisNat` somewhere. As we explained above, the top-level loop strategy first constructs the case distinction proof steps. So, when considering the structural distinction node for `Plus`, we will not directly construct a lemma application step. Instead, the top-level loop strategy will make sure that function calls seen during the construction of the case distinction steps are passed on along the generated proof edges. This is what we see in the excerpt of the generated proof graph in the lower part of Figure 6.8, highlighted in orange: We propagate the information that `isNV` was called in a top-level case along the generated proof graph.

When the top-level loop strategy arrives at leaves in the ACG being traversed, we created all the necessary case distinction steps and are considering individual cases. This is the point where we may consider to add lemma application steps. Consider Figure 6.9, which focus on the cases within the `Plus` case that we omitted in Figure 6.8. Again, the upper part of the figure depicts an excerpt of the ACG of `reduce` for these cases. The lower part of the figure visualizes an excerpt of the generated proof graph, this time for the preservation side of the generated proof: At the top of this excerpt, in black, we see the proof steps and sub-obligations that were constructed for the boolean case distinctions.

The interesting sub-case here is the third black box from the top of the proof graph excerpt, with the premises `isNV(t2)` and `isNV(t1)`. This obligation corresponds to the case where both arguments to the `Plus` constructor are numerical values. We

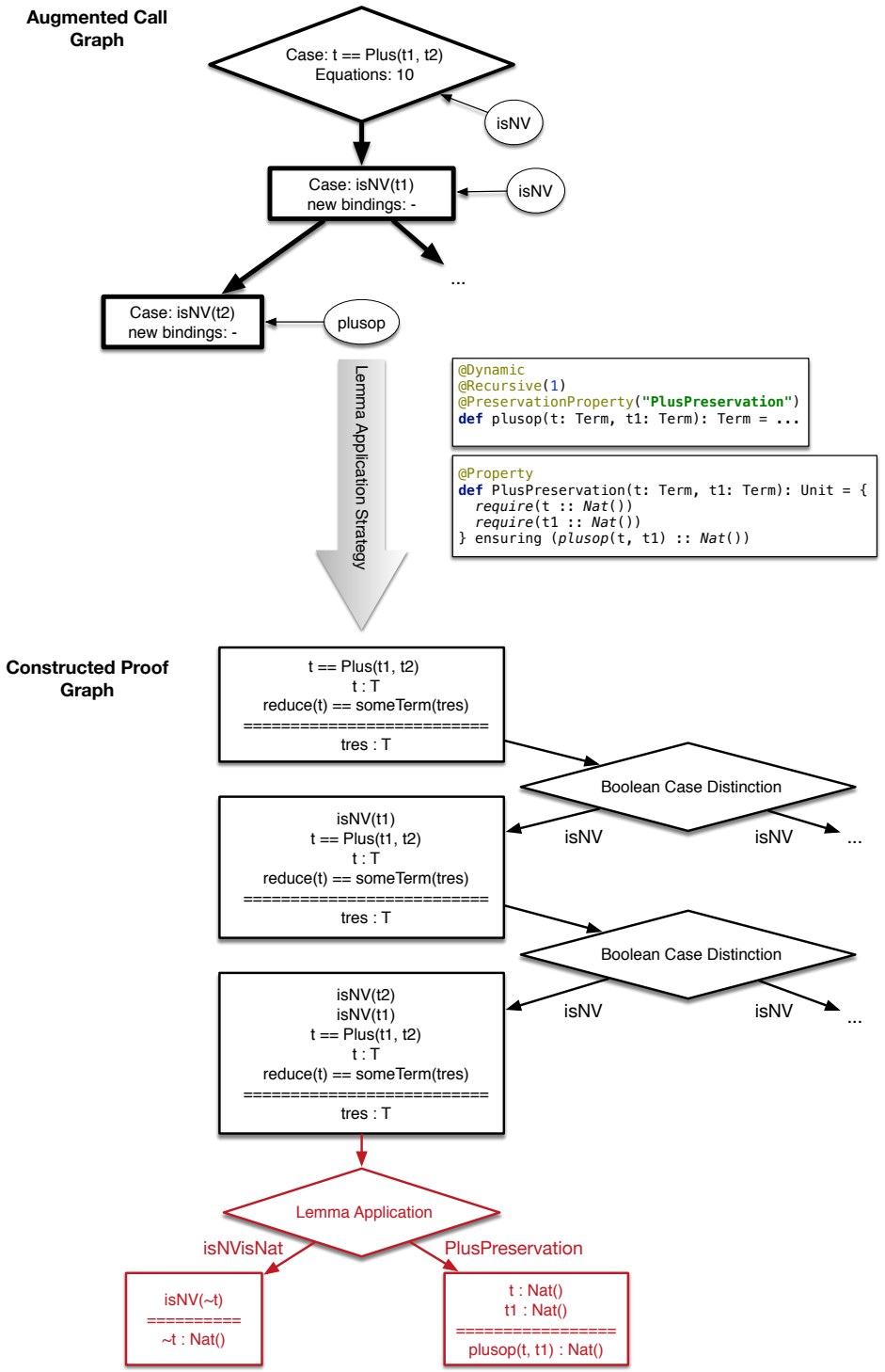


Figure 6.9.: Example of generating lemma application step within the `Plus` case of the preservation proof for typed arithmetic expressions

need to prove that in this case, the `Plus` term reduces to a values that has exactly the type `T` of the original term (the, `T = Nat`). The top-level loop strategy encounters the following situation at this point: First, the corresponding boolean distinction node in the ACG (Case: `isNV(t2)`) does not have any children. Secondly, a function call to `isNV` was propagated from some upper case during proof graph generated. Thirdly, the ACG node in question has itself a function call parent, representing a call to `plusop`, which is the function that actually performs the addition of two numerical values. So the top-level loop strategy will apply a domain-specific lemma application strategy.

The lemma application strategy will inspect the function calls that are present in the node. In the sub-case we are considering, these are `isNV` and `plusop`. The lemma application strategy will then query the collection of domain-specific knowledge to find out whether and which properties are connected to these functions. It first finds `isNVisNat`, which we explained above and showed in Figure 6.8. Furthermore, it finds `PlusPreservation`. We see this property in the lower box right of the grey arrow in the middle of 6.9. Intuitively, the property states that if the two arguments for `plusop` are both of type `Nat`, then the corresponding result of `plusop` will also be of type `Nat`. Hence, the strategy adds the proof step and sub-obligations to the proof graph as highlighted in red at the bottom of Figure 6.9.

When verifying this lemma application step with an external ATP such as Vampire, our encoding strategies will simply add the lemmas `isNVisNat` and `PlusPreservation` as axioms to the proof problem generated. In this particular case, Vampire solves the resulting proof problem less than 2 seconds.

Not every function call actually has to result in the generation of a lemma application step. If a function corresponding to a call does not have any property attached, the lemma application strategy will ignore this function. This is for example the case for the auxiliary function `getTerm`, for which a call appears in the ACG at the top of Figure 6.8. Most importantly, our lemma application strategy will ignore recursive calls. Recursive function calls indicate that at this point, we probably have to apply one or more induction hypotheses. Our strategies propagate the induction hypotheses generated during the top-level structural induction down the proof graph. They will be included in any leaf problem during verification.

Proving auxiliary lemmas At this stage, we added sub-obligations without any proof steps attached for auxiliary lemmas to the generated proof graph, connected to the remaining obligations via lemma application steps. Some lemma obligations, such as `isNVisNat`, are connected to multiple sub-obligations. Hence, our proof graph now became an actual directed acyclic graph.

Next, we need to also generate proof steps and sub-obligations for the obligations of the auxiliary lemmas, so that we are able to verify them as well. Our top-level loop strategy now considers each obligation connected to a lemma application step and finally earns the name we have given it: The strategy loops. It calls itself again, but now for the function for which each lemma application was added. For the obligation `isNVisNat`, it creates an ACG for function `isNV`, and for obligation `PlusPreservation`, it creates an ACG for function `plusop`. Then, the top-level

loop strategy repeats exactly the steps that we described above for the `reduce` function, this time with the newly generated ACGs. In these two particular cases that we consider, the functions `isNV` and `plusop` have only small ACGs, so that only applying the top-level structural induction strategy is necessary.

The top-level loop strategy stops as soon as there are no leaves left that have no proof step attached.

Verification of example At the end, our proof strategies generated a proof graph with 102 proof steps for our example proof of progress and preservation of typed arithmetic expressions. We emphasize again that this generated proof graph represents a recommendation for an overall proof structure. For the actual verification of this proof graph, we need to verify each proof step in the graph by calling external verifiers. For this, VeriTaS automatically encodes the proof problems to formats understood by existing ATPs and SMT solvers (see Chapter 5).

If we apply Vampire 4.1 on all proof steps that are not structural induction steps (which Vampire, being a first-order theorem prover, cannot verify) and a custom small verifier for induction schemes on the structural induction steps, we can automatically verify all of these steps except for 5 steps. Especially, Vampire 4.1 verifies all proof steps within the preservation part of the proof graph. The remaining 5 steps from the progress proof require manual inspection and refinement by an end user.

For completeness, we note that we have to make some minor additions to the original specification of ScalaSPL that we presented in Subsection 5.1.2 in order to arrive at this verification state: Firstly, we need to add the domain-specific annotations and the two auxiliary lemmas that we presented in the description above. Secondly, we need to add type-inversion axioms which allow us to "turn around" a typing rule and infer a rule's premises when having a term that matches the conclusion of the rule. This latter step is similar to what we have to do in different existing interactive theorem provers in order to "turn around" an inference rule. For example, in Isabelle/HOL we have to add a manual command that explicitly instructs Isabelle to create a specific inversion rule. Appendix A.1 contains the full ScalaSPL specification of our running example of typed arithmetic expressions including these additions.

6.4.2. General Patterns in Type Soundness Proofs

Figure 6.10 summarizes how the top-level loop strategy that we introduced by example in the previous subsection operates. In this section, we explain the individual general patterns that our proof strategies for our target verification domain apply.

The first meta-observation that we arrived at when developing our top-level loop strategy is that we may actually apply the same general strategy for progress and preservation proofs, as well as for the proofs of the necessary auxiliary lemmas: The high-level steps that we have to generate are always the same. There is some room for refining the individual strategies when generating lemma application steps in order to generate more "precise" proof steps. However, in our examples such

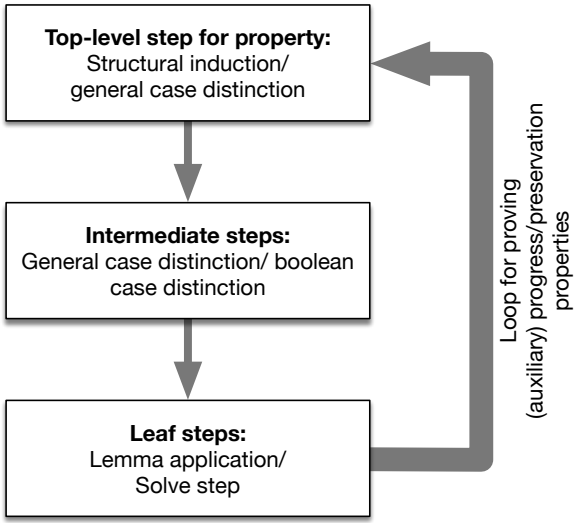


Figure 6.10.: Top-level loop strategy for generating proof steps for progress and preservation steps, including proofs of auxiliary lemmas

a refinement did not make much difference for the overall success of our proof strategies. We will discuss this point below.

Both the progress and the preservation property of a type system are always properties about a top-level small-step reduction function. Hence, the starting point of our top-level loop strategy always is a small-step reduction function that describes the dynamic semantics of a language. We assume that this function is called "reduce", marked as `@Dynamic` and as `@Recursive(i)` in position `i` of the expression argument.

The very first step of our top-level loop strategy is to generate the ACG for the function the strategy investigates. At the beginning, this is the top-level `reduce` function. For subsequent calls to the top-level loop strategy during the generation of the proof graph, the function to investigate may become a dynamic auxiliary function of the language for which auxiliary properties need to be proven.

The top-level loop strategy then traverses this ACG and constructs a proof graph according to which ACG nodes it encounters and by querying the collection of domain-specific knowledge at appropriate points.

Top-level step The top-level loop strategy decides which top-level step to construct according to the root of the structural distinction sub-tree within the ACG in question:

- If the function in question is marked as `@Recursive`, we apply a structural induction strategy, on the variable at the position argument of the `@Recursive`-annotation.

- If the function in question is not marked as `@Recursive`, we inspect the children of the considered structural distinction node and apply a general case distinction strategy.

General progress and preservation proofs in the literature start with an induction on a typing derivation (see Pierce [Pie02]). In this thesis, we consider only syntax-directed type systems (see Chapter 3). For such a type system, induction on a typing derivation and structural induction on an expression variable is equivalent. Hence, our pattern for the top-level step suffices for our target verification domain.

Intermediate steps After a structural induction on the top-level reduction function, our top-level loop strategy proceeds by distinguishing the different sub-cases present in the reduction semantics for each case:

- If the children of the current ACG node are structural distinction nodes, we apply a general case distinction strategy.
- If the children of the current ACG node are boolean distinction nodes, we apply a boolean case distinction strategy.

If the top-level loop strategy encounters a function call node during the traversal of the ACG, it propagates the call along the generated proof edges to be considered later, when generating leaf steps.

Leaf steps When the top-level loop strategy encounters a leaf within the sub-tree of structural and boolean distinction nodes of the considered ACG, it decides whether to apply a lemma application step and with which lemmas. For this, the strategy considers the function calls that were propagated along the proof edges as well as function call parents from the current ACG node. The strategy consults the collection of domain-specific knowledge in order to find any properties that are connected to the functions in question.

- If there are no function calls or no properties attached to present function calls, we apply the default Solve tactic.
- If there are function calls with properties attached, we apply a lemma application strategy with all the properties we find.

Loop After having generated the leaf steps, the top-level loop strategy looks for obligations in the generated proof graph that do not yet have proof steps attached. Such obligations are, at this point, the newly added obligations for auxiliary lemmas. Then, for each auxiliary lemma, the top-level loop strategy calls itself, with the function which is connected to the auxiliary lemma in question.

Lemma application patterns Our generation technique for lemma application steps is an over approximation. The exact patterns within the basic progress and preservation proofs we consider in this thesis are as follows:

- **Progress pattern**: A sub-case calls a function `f` that can *fail* (i.e. does not return a concrete value for a certain argument combination). The sub-case will return a valid result itself if `f` produces a result, otherwise the sub-case will try something else or fail itself. The sub-case may or may not use the result of `f` to construct its result. In this case, we need to apply a "progress property" for function `f`: Assuming some static conditions (arising from typing) hold for the function's arguments, it follows that the application of `f` to these arguments does not fail. Pacak describes this pattern and how exactly it looks like within an ACG in more details in his thesis [Pac18].
- **Preservation pattern**: A sub-case calls a function `f` that *transforms* an expression or part of an expression. The result of this call is used for constructing the result of the sub-case. In this case, we need to apply a "preservation property" for function `f`: Assuming some static conditions (arising from typing) hold for the function's arguments, and also assuming that calling `f` for these arguments returns a result (in case `f` also is a function that can fail), it follows that the same static conditions (or a subset of them) have to hold for the result of the function.
- **Other properties**: If a function call within a sub-case falls in neither of the categories above, we may require some other auxiliary property for it. A concrete example of such a property is the property `isNVisNat` from our example from above.

These exact pattern motivate the names of our two domain-specific annotations `@ProgressProperty` and `@PreservationProperty`) that link properties to functions. These names provide some intuition as to which properties may be required for the generation of progress and preservation proofs.

One may be tempted to think that a proof of progress will only ever require progress properties, and a proof of preservation will only ever require preservation properties. However, this is not true in general, since sub-cases may contain multiple function calls that use the arguments of each other, so that for example a failable function `g` uses the result of a function `f` in its arguments. In this case, even when we are proving progress, we will first have to apply a preservation property on `f` so that we can actually satisfy the premises of a progress property for `g`. We will see a concrete example for these more complex cases in Chapter 7.

It would be possible to exactly determine from the ACGs which of the described patterns appear within a sub-case. However, since we use powerful ATPs to prove the individual obligations that we generate, we approximate the pattern for lemma application as follows stated above: We simply always add all progress and preservation properties that we find for a function to a generated lemma application step.

Using this simplified proof pattern for lemma application, we might end up introducing unnecessary lemma applications into the generated proof graph. At least for small proofs, however, these unnecessary lemma applications are unlikely to hurt: ATPs called for the corresponding proof steps within a proof graph may simply ignore the unnecessary lemmas. In all example specifications that we consider in this thesis, the presence of unnecessary lemmas in the generated proof problems did not influence the ATPs we used. If indeed a proof step cannot be proven by external verifiers, users may attempt to remove unnecessary lemmas manually from the generated proof graph.

6.4.3. Proof Strategies for Other Verification Domains

We summarize a possible series of general steps for implementing proof strategies in VeriTaS for arbitrary verification domains. Some of the steps we describe reuse implementations of components that we implemented for our target verification domain.

1. Define a collection of domain-specific knowledge on your verification domain. Domain-specific knowledge may for example be a classification of functions in specifications, specific properties of functions, and links between properties in a specification and functions.

2. Generate such a collection from an input specification in your domain. This generation may either be entirely automatic, or you may provide user annotation constructs for specifications from which the necessary information will be collected.

3. Check whether the information given by an augmented call graph will be enough to automatically generate proof steps. We expect the information to be sufficient for a number of verification domains within general program verification. If something is missing, extend the augmented call graphs we presented in this chapter accordingly. If you work in an entirely different verification domain (e.g., the verification of cryptographic protocols), you may need to redefine ACGs.

4. If you use your own format for input specifications, implement a builder that constructs an ACG from a specification in your format. Furthermore, implement your own `SpecEnquirer` for your format if you want to be able to reuse existing tactics and proof strategies. Alternatively, you may implement a translation from your input format to SPL or ScalaSPL in order to re-use the existing ACG builder and `SpecEnquirer`. If you reuse SPL or ScalaSPL, you can skip this step.

5. Implement your own proof strategies (general as well as domain-specific ones) by triggering the generation of ACGs for appropriate functions, traversing them, and adding proof steps and sub-obligations to the proof graph you are constructing according to the information in the ACG nodes. For this step,

you may for example discover patterns in ACGs for input specification you consider and implement one proof strategy per such pattern. You may also be able to reuse some of the tactics and proof strategies that we implemented for our target verification domain.

As we mentioned above, our top-level loop strategy actually works well for a number of different proofs: for top-level progress and preservation proofs as well as for the proofs of auxiliary properties needed along the way. Hence, we expect that great parts of this strategy may be re-used and adapted for other verification domains in the area of program verification. To adapt our top-level loop strategy for other verification domains one probably has to adapt the initialization strategy, the generation of the top-level induction step (add different induction techniques, depending on the verification domain), and the lemma application strategy.

6.5. Summary

We presented our approach for automatically generating proof graphs for our target verification domain. Our proof strategies use two components that serve as intermediate data structures for capturing domain-specific aspects of type system specifications and for abstracting over parts of specifications that are not relevant for proof generation: a collection of domain-specific knowledge and what we named augmented call graphs.

For the first component, we provide domain-specific user annotations in ScalaSPL for type system specifications. These annotations allow end users to for example categorize functions within the specification as either static or dynamic and to link certain properties to functions. We automatically collect the information given by these annotations into a Scala trait.

Augmented call graphs employ concepts from both control-flow graphs and call graphs. They hierarchically group the different control-flow steps within the definition of a function. Furthermore, they contain the information which function in a specification calls which other function and in which control branch exactly the result of a call is used.

Our proof strategies traverse augmented call graphs for dynamic functions within a type system specification and generate proof steps and sub-obligations in the constructed proof graph. For this generation, they exploit the information within the given nodes in the augmented call graphs and query the domain-specific knowledge.

We also described how the proof strategies that we implemented for our target verification domain may be reused for other verification domains: Most of the strategies we implemented turned out to be fairly general in their overall nature, and therefore suitable for reusal.

Chapter

7

Case Studies

We present two case studies for studying the effectivity of our automated proof strategies from Chapter 6 and for assessing the overall usability of VeriTaS for formal verification: the subset of typed SQL from Chapter 3, and another DSL for creating questionnaires, called QL. We first describe both of the case studies and their results separately (Sections 7.1 and 7.2): For each study, we first present their ScalaSPL specifications within VeriTaS. Next, we show how the automated proof strategies from Chapter 6 generate proof graphs for both case studies, with first intermediate verification results. Finally, we evaluate in detail how many and which of the proof steps in the generated graphs may be verified automatically by the ATPs connected to VeriTaS and which steps require further user interaction. After having presented both case studies, we discuss our results and compare the two case studies to each other. Furthermore, we argue how and why we may generalize our observations from our two case studies to other DSLs within our target domain. Finally, we compare the results from this chapter to our observations from Chapter 3.

We find that our proof strategies indeed break down progress and preservation proofs into proof steps that automated provers can, for the most part, prove automatically. In our two cases studies, only a small number of individual proof steps remain which the provers we used cannot directly verify, due to their heuristic nature. Compared to the verification systems we looked at in detail in Chapter 3, our instantiation of VeriTaS indeed lowers the user effort for obtaining mechanized progress and preservation proofs.

Remark 7.1. The author of this thesis co-published the two specifications of the case studies presented within this chapter in a journal paper at the "Science of Computer Programming" journal in 2018 [Gre+18a]. Earlier versions of the specification of typed SQL were published at the international conferences Onward! in 2015 [Gre+15] and "Principles and Practice of Declarative Programming (PPDP)" in 2016 [Gre+16]. The remainder of this chapter, i.e. using these two specifications for evaluating our proof strategies for type soundness proofs, is unpublished. ◊

7.1. A Subset of Typed SQL

As a first case study, we use the typed subset of SQL that we introduced at the beginning of this thesis in Section 3.1.3 and that we studied in detail in Chapter 3. There, we formalized a type system for this language and developed its progress and preservation proof in Isabelle/HOL and in Dafny. We now specify SQL's type system in ScalaSPL within VeriTaS. We stay as close as possible to the specifications in Isabelle/HOL and Dafny presented in Chapter 3. Occasionally, minor derivations from these specifications become necessary to accommodate the features available in ScalaSPL and VeriTaS, which we will name in the following text.

7.1.1. Specification of SQL in ScalaSPL

We first focus on presenting the top-level parts of the specification of typed SQL in ScalaSPL. We will leave out domain-specific annotations for auxiliary functions in this subsection. We will demonstrate in the next subsection how intermediate generation results for such partially annotated specifications will look like and ultimately add the remaining user annotations there to obtain refine the generated proof graph.

We start with encoding basic data structures such as rows and tables. The encoding of these data structures in ScalaSPL is conceptually similar to the encoding we presented in Chapter 3. However, one major difference to the specifications in Isabelle/HOL and Dafny is that ScalaSPL does not support Scala's standard library. In particular, there is no built-in generic list or array construct. Also, ScalaSPL does not support any kind of type parameters for specifying higher-level data structures. Adding these features would be conceptually straightforward, but require additional engineering effort that was not in the scope of this thesis.

To encode things like lists and arrays in the core implementation of ScalaSPL in this thesis, we therefore need to explicitly introduce "hand-made" recursive list constructs for each separate list type. For example, we specify the basic data structures for tables as in Listing 7.1:

```
1  // list of attribute names
2  sealed trait AttrL extends Expression
3  case class aempty() extends AttrL
4  case class acons(hd: Name, tl: AttrL) extends AttrL
5
6  // Value for a field (underspecified)
7  trait Val extends Expression
8
9  // table row, list of field values (with at least one cell/field per construction!)
10 sealed trait Row extends Expression
11 case class rempty() extends Row
12 case class rcons(v: Val, r: Row) extends Row
13
14 // table matrix (list of rows), without "header" (attribute list)
15 sealed trait RawTable extends Expression
16 case class tempty() extends RawTable
```

```
case class tcons(r: Row, rt: RawTable) extends RawTable

// full table with "header" (attribute list)
sealed trait Table extends Expression
case class table(a: AttrL, rt: RawTable) extends Table
```

Listing 7.1: Basic data structures for tables in ScalaSPL

Note that we let all of the types introduced above extend trait `Expression` provided by trait `ScalaSPLSpecification` to mark them as part of the domain for expressions of our language. This is required for being able to use the syntactic sugar for typing judgments whose use for specifying the type system for SQL we will see below. Similarly, the types that model table types have to extend trait `Type` and the type that models table type context has to extend trait `Context`.

We also introduce separate list constructs such as the ones shown above when specifying table stores, table types, and table type contexts.

Similarly, ScalaSPL does not provide a generic option datatype, so we need to introduce one custom option type wherever necessary. For example, Listing 7.2 shows the type we introduce to model queries along with the corresponding option type and some utility functions:

```
sealed trait Query extends Expression
case class tvalue(t: Table) extends Query
case class selectFromWhere(s: Select, name: Name, pred: Pred) extends Query
case class Union(q1: Query, q2: Query) extends Query
case class Intersection(q1: Query, q2: Query) extends Query
case class Difference(q1: Query, q2: Query) extends Query

def isValue(q: Query): Boolean = q match {
  case tvalue(_) => true
  case selectFromWhere(_, _, _) => false
  case Union(_, _) => false
  case Intersection(_, _) => false
  case Difference(_, _) => false
}

sealed trait OptQuery
case class noQuery() extends OptQuery
case class someQuery(q: Query) extends OptQuery

def isSomeQuery(oq: OptQuery): Boolean = oq match {
  case noQuery() => false
  case someQuery(_) => true
}

@Partial
def getQuery(oq: OptQuery): Query = oq match {
  case someQuery(q) => q
}
```

Listing 7.2: Queries and option type for queries in ScalaSPL

The annotation `@Partial` for function `getQuery` is crucial to ensure that the axiomatization of this function in TPTP will not generate an inversion axiom, which would introduce inconsistencies since `getQuery` is deliberately undefined for `noQuery`.

Next, we show an excerpt of the fully annotated specification of the top-level reduction function `reduce` for our subset of typed SQL:

```
1  @Dynamic
2  @ProgressProperty("Progress")
3  @PreservationProperty("Preservation")
4  @Recursive(0)
5  def reduce(query: Query, tst: TStore): OptQuery = (query, tst) match {
6    case (tvalue(_), _) => noQuery()
7    case (selectFromWhere(s, n, p), ts) =>
8      val maybeTable = lookupStore(n, ts)
9      if (isSomeTable(maybeTable)) {
10       val filtered = filterTable(getTable(maybeTable), p)
11       val maybeSelected = projectTable(s, filtered)
12       if (isSomeTable(maybeSelected))
13         someQuery(tvalue(getTable(maybeSelected)))
14       else noQuery()
15     } else
16       noQuery()
17   case (Union(tvalue(t1), tvalue(t2)), ts) =>
18     someQuery(tvalue(table(getAttrL(t1), rawUnion(getRaw(t1), getRaw(t2)))))
19   case (Union(tvalue(t), q2), ts) =>
20     val q2reduce = reduce(q2, ts)
21     if (isSomeQuery(q2reduce))
22       someQuery(Union(tvalue(t), getQuery(q2reduce)))
23     else
24       noQuery()
25   case (Union(q1, q2), ts) =>
26     val q1reduce = reduce(q1, ts)
27     if (isSomeQuery(q1reduce))
28       someQuery(Union(getQuery(q1reduce), q2))
29     else
30       noQuery()
31    ...
```

Listing 7.3: Excerpt of semantics for typed SQL in ScalaSPL

This specification is largely similar to the specifications of the same function within Isabelle/HOL and Dafny that we presented in Chapter 3. We annotate the function firstly as `@Dynamic`, since it is part of the specification of the dynamic semantics for our subset of SQL. The top-level proof strategy for the generation of the progress/preservation proof graph requires this annotation to correctly look the definition of `reduce` up in the collected domain-specific knowledge. Next, we link the top-level progress and preservation theorem to `reduce` with the next two annotations (lines 2 and 3). These two annotations are again important for the top-level proof strategy which looks these two properties up in the specification

and adds them as root nodes to the generated proof graph. Finally, we annotate `reduce` as `@Recursive` in its first argument, which will instruct the top-level strategy to apply a structural induction on this argument as first proof step for proving properties about `reduce`.

We omit showing the ScalaSPL definitions of the necessary auxiliary functions at this point. They are also largely equivalent to the corresponding Isabelle/HOL and Dafny specifications. In this first version of the specification of typed SQL, we leave the auxiliary functions unannotated, which will trigger the generation of an incomplete proof graph.

Next, we specify the type system of our language in ScalaSPL. Listing 7.4 shows the typing rule as well as its inversion rule for `selectFromWhere`.

```
1  @Axiom
2  def TSelectFromWhere(tn: Name, TTC: TTContext, TT: TType, p: Pred, sel: Select, TTr:
       TType): Unit = {
3    require(lookupContext(tn, TTC) == someTType(TT))
4    require(tcheckPred(p, TT))
5    require(projectType(sel, TT) == someTType(TTr))
6  } ensuring(TTC |- selectFromWhere(sel, tn, p) :: TTr)
7
8  @Axiom
9  def TSelectFromWhere_inv(tn: Name, TTC: TTContext, p: Pred, sel: Select, TTr: TType):
       Unit = {
10   require(TTC |- selectFromWhere(sel, tn, p) :: TTr)
11 } ensuring(exists((TT: TType) => lookupContext(tn, TTC) == someTType(TT) &&
12     tcheckPred(p, TT) &&
13     projectType(sel, TT) == someTType(TTr)))
```

Listing 7.4: Excerpt of type system and type inversion axioms for typed SQL in ScalaSPL

Note that both rules require the `@Axiom` annotation, otherwise the internal translation to SPL will complain that the bodies of the functions `TSelectFromWhere` and `TSelectFromWhere_inv` do not have the expected shape. In both rules, we use the syntactic sugar for typing judgments `_ ⊢ _ :: _` that ScalaSPL provides. As opposed to the previously shown specifications of typed SQL, in ScalaSPL we explicitly have to add inversion axioms for a typing rule that will allow provers to assume the premises of a typing rule from its conclusion. Conceptually, this manual addition is not different from what other verification systems require. For instance, in Isabelle/HOL, one also has to manually trigger the generation of such inversion rules via command `inductive_cases`. Note that in our Dafny specification of typed SQL, we do not need to specify any inversion axioms since we had to specify the type system as a function predicate instead of as a set of inference rules.

Finally, we show the top-level progress and preservation theorems for typed SQL in ScalaSPL in Listing 7.5, which are equivalent to the corresponding theorem in our Isabelle/HOL and Dafny specification of typed SQL.

```
@Property
def Progress(ts: TStore, ttc: TTContext, q: Query): Unit = {
  require(storeContextConsistent(ts, ttc))
  require(!isValue(q))
  require(exists((tt1: TType) => ttc |− q :: tt1))
} ensuring exists((qr: Query) => reduce(q, ts) == someQuery(qr))

@Property
def Preservation(ttc: TTContext, ts: TStore, q: Query, qr: Query, tt: TType): Unit = {
  require(storeContextConsistent(ts, ttc))
  require(ttc |− q :: tt)
  require(reduce(q, ts) == someQuery(qr))
} ensuring(ttc |− qr :: tt)
```

Listing 7.5: Top-level progress and preservation theorems for typed SQL in ScalaSPL

Note that both definitions need to be annotated as `@Property` so that the collection of domain-specific knowledge can correctly link them to the `reduce` function.

7.1.2. Automated Generation of Proof Graph

Top-level proof steps We first study the proof graph that our automated proof strategies from Chapter 6 generate for the top-level theorems if we just use the domain-specific annotations that we showed in the previous subsection.

Figure 7.1 shows an excerpt of the progress part of the generated proof graph for the top-level progress and preservation proof. The full graph also contains the preservation graph, which is structurally equal to the progress part, since the automated generation of both graphs is triggered by the definition structure of the `reduce` function. That is, initially the full proof graph is actually a forest consisting of two trees.

We applied different external automated provers on the proof steps of the proof graph from Figure 7.1. We obtain the best (intermediate) result for Vampire 4.3.0 in casc mode with a timeout of 120 seconds. Figure 7.1 visualizes the result of applying Vampire 4.3.0 to all but the top-level structural induction proof steps via node colors: Filled green nodes indicate that Vampire could find a proof, unfilled red nodes indicate that the search for a proof was unsuccessful within the given timeout. Note that unsuccessful verifications of individual proof steps get propagated to the top of the parent nodes. For example, `PreservationUnion` is marked red since the sub-case `PreservationUniontvaluetvalue` could not be verified.

For the top-level structural induction step, we apply a simple custom verifier that checks the basic validity of the applied induction scheme and returns the scheme as evidence for users to inspect. Our proof strategies generate the individual case names automatically by appending constructor and/or function names that appear within case expressions. For example, the name `PreservationUniontvaluetvalue` stands for the case for the `Union` constructor with a `tvalue` expression for both of its arguments within the `reduce` function.

We can see in the preliminary proof graph that Vampire proves many sub-cases and intermediate proof steps (case distinctions) without any problem. In the graph, we

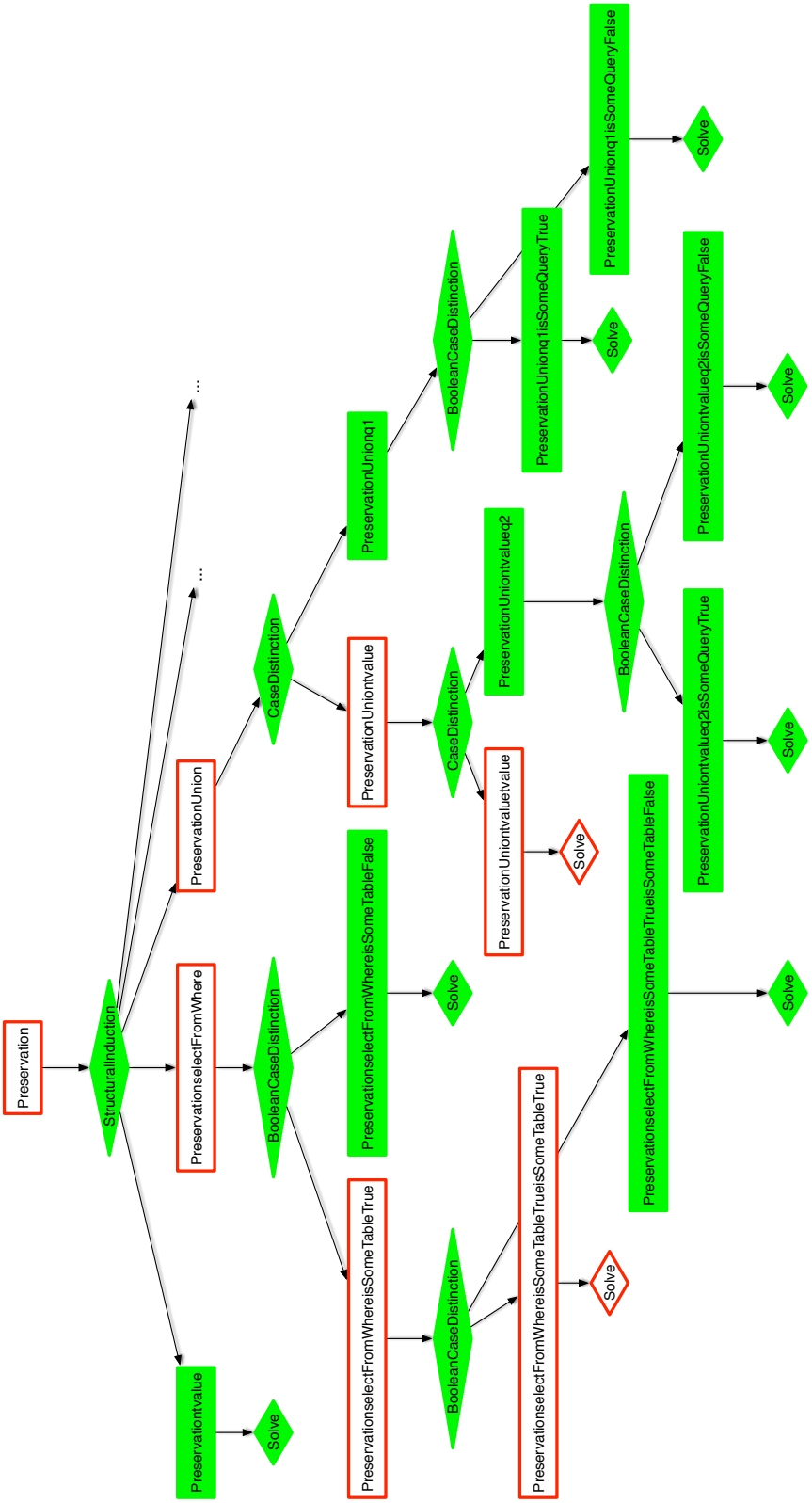


Figure 7.1.: Excerpt of top-level proof graph for preservation proof of typed subset of SQL, with intermediate verification state

omitted the parts for `PreservationIntersection` and `PreservationDifference`, since they are structurally equivalent to the sub-tree for `PreservationUnion`, also regarding their intermediate verification status. We also note that the main reason why not everything is marked as verified here is due to the unsuccessful verification of the cases `PreservationUniontvaluetvalue` (resp. the corresponding omitted cases within `PreservationIntersection` and `PreservationDifference`) and `PreservationselectFromWhereisSomeTableTrueisSomeTableTrue`.

Adding auxiliary lemmas via annotations We may inspect the unverified sub-goals more closely by instructing VeriTaS to pretty-print the generated problems in ScalaSPL. Here, this inspection reveals that in all corresponding cases of the `reduce` function, the function definition calls auxiliary functions, e.g. function `rawUnion` with the `Union(tvalue(...), tvalue(...))` case. That is, at these points we will need to propagate the progress and preservation property to the auxiliary functions used (see Section 6.4.2).

It is clear that the low-level automated theorem provers will not be able to deduce the necessary auxiliary lemmas by themselves. Hence, users needs to specify auxiliary lemmas about the progress and/or preservation properties of the auxiliary functions used. Then, they may link these properties to the corresponding auxiliary functions via the user annotations we provide (see Section 6.2). With these annotations, the automated proof strategies will add the corresponding lemmas and proof steps for them to the generated proof graph at the correct points.

For example, we add a preservation property for `rawUnion` by adding the following to the ScalaSPL specification of typed SQL:

```
1  @Dynamic
2  @PreservationProperty("rawUnionPreservesWellTypedRaw")
3  @Recursive(0)
4  def rawUnion(rtab1: RawTable, rtab2: RawTable): RawTable = ...
5  ...
6  @Property
7  def rawUnionPreservesWellTypedRaw(rt: RawTable, rt1: RawTable, result: RawTable, tt:
       TType): Unit = {
8    require(welltypedRawtable(tt, rt))
9    require(welltypedRawtable(tt, rt1))
10   require(rawUnion(rt, rt1) == result)
11 } ensuring welltypedRawtable(tt, result)
```

Listing 7.6: Adding an auxiliary preservation property for `rawUnion`

The auxiliary preservation property for `rawUnion` (lines 6 to 11 in Listing 7.6) states that upon receiving two well-typed argument tables (with the same table type `tt`), the result of `rawUnion` for these tables is also well-typed (again with table type `tt`). The annotation in line 2 links this property as preservation property to `rawUnion`. Note that we do not have to add a progress property for `rawUnion` since this function is specified so that it always returns some result.

Figure 7.2 shows the excerpt for induction case `PreservationUnion` that the strategies generate with the addition of the information in Listing 7.6. In the figure,

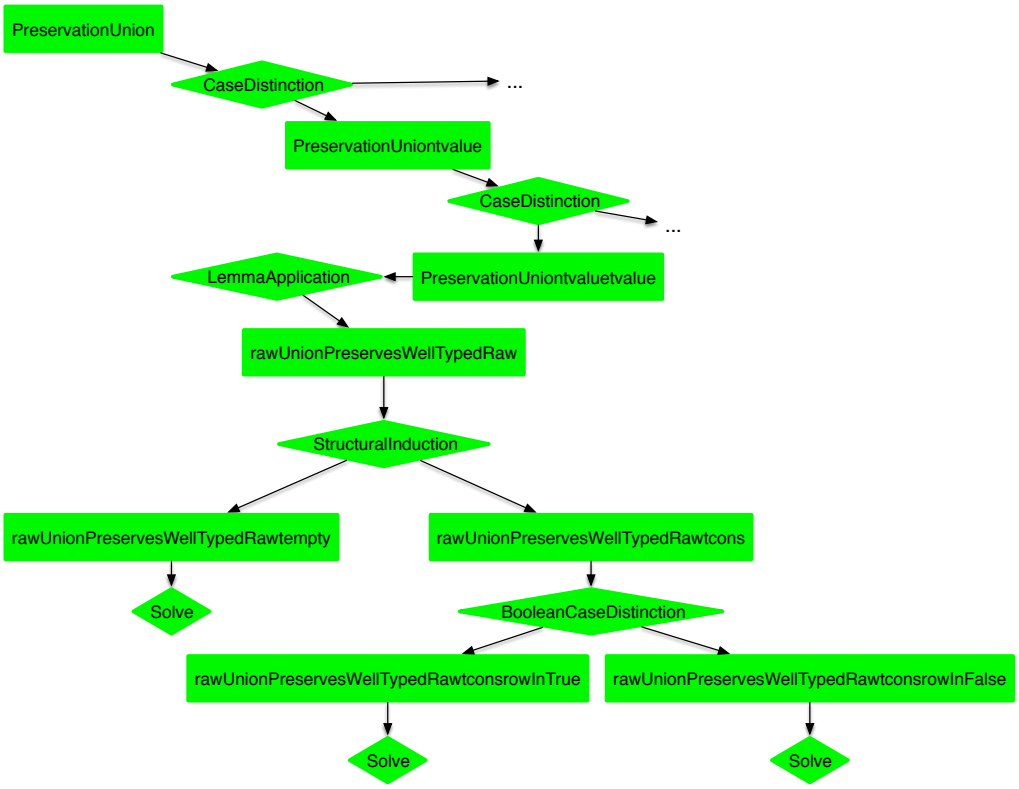


Figure 7.2.: Sub-proof graph from preservation proof of typed subset of SQL: Refined proof steps for the sub-case of the `Union` induction case

we omitted the parts for the cases we already showed in Figure 7.1. For the previously incomplete sub-case `PreservationUniontvaluetvalue`, we now obtain a lemma application node linking to the auxiliary lemma `rawUnionPreservesWellTypedRaw`. Furthermore, the automated strategies add proof steps for this auxiliary lemma according to the generated ACG of `rawUnion`. When applying Vampire 4.3.0 to the newly generated proof steps, all of them, including the lemma application step, can be verified, thus verifying the `PreservationUnion` case. For the induction cases `PreservationIntersection` and `PreservationDifference`, we apply equivalent steps which also result in equivalent sub-proof graphs where all steps are verified.

We now consider the inconclusive sub-case in `PreservationselectFromWhere`: The corresponding case within the `reduce` function depends on the results of 3 relevant auxiliary functions: `lookupStore`, `filterTable`, and `projectTable`. In this sub-case, we have a concrete example of a case in a preservation proof where we do not only need auxiliary preservation properties, but also auxiliary progress properties. Function `filterTable` cannot fail and hence will "only" require a preservation property. But both the functions `lookupStore` and `projectTable` may also fail, hence the proof might require *both* a progress and a preservation property.

From our proofs of the specification of typed SQL using other verification tools (see

Chapter 3) we know that we will in fact need a progress property for `lookupStore`, but not for `projectTable` (see for example the Dafny proof of the `selectFromWhere` case for preservation in Listing 3.29). Still, from the perspective of a user who has not done the proof in detail, we conservatively add progress properties for *all* function calls that may fail. The necessary additions within the ScalaSPL specification look as follows:

```
@Dynamic
@ProgressProperty("successfulLookup")
@PreservationProperty("welltypedLookup")
@Recursive(1)
def lookupStore(an: Name, tst: TStore): OptTable = ...

@Dynamic
@ProgressProperty("projectTableProgress")
@PreservationProperty("projectTableWelltypedWithSelectType")
def projectTable(s: Select, tab: Table): OptTable = ...

@Dynamic
@PreservationProperty("filterPreservesType")
def filterTable(t: Table, pred: Pred): Table = ...

...

@Property
def successfulLookup(ttc: TTContext, ts: TStore, ref: Name, tt: TType): Unit = {
  require(storeContextConsistent(ts, ttc))
  require(lookupContext(ref, ttc) == someTType(tt))
} ensuring exists((t: Table) => lookupStore(ref, ts) == someTable(t))

@Property
def welltypedLookup(ttc: TTContext, ts: TStore, ref: Name, tt: TType, t: Table): Unit = {
  require(storeContextConsistent(ts, ttc))
  require(lookupContext(ref, ttc) == someTType(tt))
  require(lookupStore(ref, ts) == someTable(t))
} ensuring welltypedtable(tt, t)

@Property
def projectTableProgress(tt: TType, t: Table, s: Select, tt2: TType): Unit = {
  require(welltypedtable(tt, t))
  require(projectType(s, tt) == someTType(tt2))
} ensuring exists((t2: Table) => projectTable(s, t) == someTable(t2))

@Property
def projectTableWelltypedWithSelectType(s: Select, t: Table, t1: Table, tt: TType, tt1:
    TType): Unit = {
  require(welltypedtable(tt, t))
  require(projectType(s, tt) == someTType(tt1))
  require(projectTable(s, t) == someTable(t1))
} ensuring welltypedtable(tt1, t1)
```

```
43
44 @Property
45 def filterPreservesType(tt: TType, t: Table, result: Table, p: Pred): Unit = {
46   require(welltypedtable(tt, t))
47   require(filterTable(t, p) == result)
48 } ensuring welltypedtable(tt, result)
```

Listing 7.7: Adding auxiliary progress and preservation properties for the top-level auxiliary functions used in the `selectFromWhere` case

With these additions, the automated proof strategies append the proof steps visualized in Figure 7.3 to the overall proof graph within the corresponding sub-case for `PreservationselectFromWhere`. Note that the addition of the properties and annotations from Listing 7.7 will also cause the same lemma applications to be added within the progress part of the proof, i.e. the auxiliary lemmas will be reused within the proof graph at different places. The corresponding part within a manual progress proof requires the auxiliary properties `successfulLookup`, `welltypedLookup`, `filterPreservesType`, and `projectTableProgress`, but no preservation property for `projectTable`.

As for verifying the proof steps in the proof graph from Figure 7.3, we obtain the best results with Vampire 4.1 and a timeout of 120 seconds: We see that Vampire 4.1 verifies all proof steps for the auxiliary lemmas `successfulLookup`, `welltypedLookup`, and `filterPreservesType`. Note that for verifying the latter lemma, we added another auxiliary lemma in Figure 7.3: Function `filterTable` has only a single case that calls the auxiliary function `filterRows`. Hence we also added a corresponding preservation property for `filterRows`, which is then automatically applied to prove `filterPreservesType`.

However, some of the cases for the remaining auxiliary lemmas for `projectTable` cannot yet be proved by Vampire: These cases call an additional auxiliary function, which in turn calls several auxiliary functions. To refine these proof steps, we need to add auxiliary progress and preservation properties for these functions as well and annotate them accordingly. All properties and annotated functions are included in the full ScalaSPL specification of typed SQL in the Appendix A.

Most noteworthy in Figure 7.3 is that it contains a very hard proof step: the lemma application at the top, which refined the sub-case of `PreservationselectFromWhere`. As mentioned above, also the manual/interactive part of this proof requires applying 4 lemmas in their correct order and in the correct instantiation. Given a sufficiently high timeout (120 seconds), Vampire 4.1 actually finds a proof of this step in 92 seconds. (Note that this value may be subject to some changes, depending on the resources available for Vampire - e.g. on another machine, we had to use a timeout of 150 seconds to obtain a proof within 101 seconds.) If we inspect the proof returned by Vampire, we find that it also uses the 4 auxiliary lemmas as expected.

7.1.3. Evaluation: Verification of Proof Steps

In Table 7.1, we give an overview of the complete verification results we obtained for the typed SQL case study.

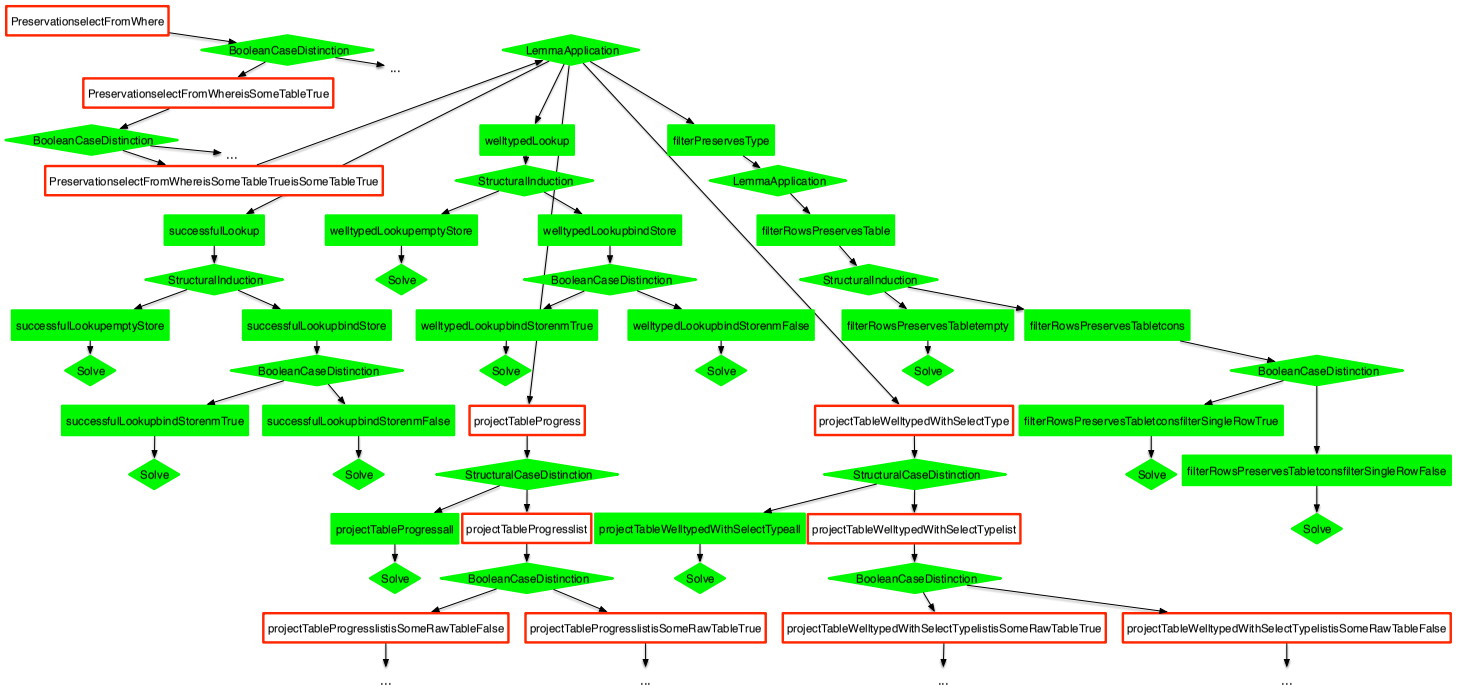


Figure 7.3.: Sub-proof graph from preservation proof of typed subset of SQL: Refined proof steps for the sub-case of the `SelectFromWhere` induction case

	Total	Proved	Inconclusive
Induction steps	22	22	0
Non-induction steps	157	149	8
Total	**179**	**171**	**8**

Table 7.1.: Overview of proof steps and their verification status within the typed SQL case study

For the fully annotated ScalaSPL specification of our subset of typed SQL, our automated proof strategies generate a proof graph with 179 individual proof steps. Out of these steps, 22 steps are top-level structural induction steps for which we cannot use the first-order ATPs and SMT solvers (2 applications for the top-level theorems, 20 for the 20 auxiliary lemmas that we added). For the structural induction steps, we use a small internal induction scheme checker, as described above.

Consistency checks First, we can also use the verifiers connected to VeriTaS to look for inconsistencies/contradictions within our specification: For this, we simply use a slight modification of our problem encoding, where we add the goal of each problem as an axiom and instruct the ATP to prove false. If the ATP cannot find a proof within a high timeout, the probability that this particular problem contains an inconsistency or a contradiction is very low. Of course, this check does not permit us to definitely exclude the possibility that specifications contain bugs. However, in practice, we observed that Vampire was typically able to find any contradictions arising from small specification mistakes within under a second.

We let Vampire 4.1 run for 300 seconds for every generated proof problem for the progress and preservation proof of SQL. Every run was inconclusive. Hence, we can safely assume that our specifications do not contain any inconsistencies and that the proofs we obtain are reasonable.

Verification results for induction cases Next, we apply provers to all 157 proof steps in the generated graph that are not structural induction steps. For all steps that are not structural induction steps, we obtain the best results with Vampire 4.1 and Vampire 4.3.0 (which internally also uses Z3), with a timeout of 120 seconds. We obtain proofs for 149 of these problems within this timeout, i.e. for roughly 95 percent of all problems.

To put the remaining 8 inconclusive problems into perspective, we analyze the nature of all problems (and notably the inconclusive ones) further, from the perspective of a domain expert who would manually prove the problems. We observe four different kinds of proof problems within our generated problems (apart from the top-level applications of structural induction schemes, which we do not send to the ATPs anyway):

- **Proof "by definition"**: In a manual proof, one would apply the axioms

generated for function and datatype definitions, as well as the axioms for typing rules and typing inversion.

- **Case distinctions (general and boolean)**: In a manual proof, one would apply the additional axioms for the sub-cases as well as some of the axioms arising from the datatype definitions in order to prove that really all cases are covered by the given axioms for the sub-cases.

- **Applications of induction hypotheses**: In a manual proof, one would apply one or more of the given induction hypotheses, along with some axioms arising from function and datatype definitions, as well as the axioms for typing rules and typing inversion in order to connect the case premises to the premise of the induction hypotheses and to connect the obtained conclusion(s) from the induction hypotheses to the conclusion of the case.

- **Applications of one or more auxiliary lemmas (and possibly induction hypotheses)**: In a manual proof, one would apply one or more of the given auxiliary lemmas, potentially also one or more of the given induction hypotheses. In addition, one might have to apply axioms arising from function and datatype definitions, as well as the axioms for typing rules and typing inversion.

The categories in the list above are roughly sorted by ascending difficulty (for a human who is doing a manual proof). Note that the ATP sometimes "uses" different facts than a human would have used, and often uses more facts than necessary. For example, there are some proof problems that do not necessarily require the application of an induction hypothesis, but the prover nevertheless used it to discover the proof. In that case, we would not consider the problem as an "application of induction hypotheses" in our categorization. Similarly, sometimes the prover may rather choose to not use a given lemma within a certain sub-case, but rather to "inline" the proof of the case of the lemma that is needed. This different behavior is due to the nature of ATPs, who do not, like a human, "apply" lemmas or definitions. Rather, they interpret a problem as a large set of clauses on which they apply calculi such as binary resolution and superposition.

Nevertheless, we decided to categorize the generated proof problems from the perspective of a human, so that users may gain a better understanding of which problems they can in general expect an ATP to solve and which might need additional user interaction.

Table 7.2 gives an overview of the problem distribution within the typed SQL case study. As we can see, most problems are case distinctions, closely followed by applications of induction hypotheses. Most of the inconclusive problems in this case study appear in the category "lemma applications". Ultimately, these problems require manual interaction. We discuss this further in Section 7.3.

Problem Category	Proved	Inconclusive	%
Proof by definition	38	-	0
Case distinctions	50	-	0
IH applications	43	2	4.6
Lemma applications	18	6	33.3

Table 7.2.: Categorization of generated proof problems for subset of typed SQL, together with verification state

7.2. A DSL for Questionnaires (QL)

We add a second, different case study to strengthen the results of this thesis and to demonstrate that our proof strategies for progress and preservation proofs are applicable to several type system specifications. In Section 7.3, we will motivate the choice our our two case studies further and compare both studies against each other.

Our second case study is a DSL that has been proposed independent and prior to our work as benchmark language to facilitate the study of language-workbench capabilities [Erd+13; Erd+15] and that has for example been used within the "Language Workbench Challenge" [Erd+13]: the "Questionnaire Language" (QL for short), a typed language for building executable questionnaires.

7.2.1. Introduction of QL

The questionnaire language we model was originally specified for the "Language Workbench Challenge" 2013, where participants could demonstrate the capabilities of their language workbench for creating a DSL. As an informal description of the language, we cite excerpts from the original specification of the challenge, which to date is not available online anymore.

Syntax of QL

> "QL consists of questions grouped in a top-level form construct. First, each question is identified by a name that at the same time represents the result of the question. In other words, the name of a question is also the variable that holds the answer. Second, a question has a label that contains the actual question text presented to the user. [...] Third, every question has a type. Finally, a question can optionally be associated to an expression: this makes the question computed.
>
> A questionnaire consists of a number of questions arranged in sequential and conditional structures, and grouping constructs. Sequential composition prescribes the order of presentation. Conditional structures associate an enabling condition to a question, in which case the question should only be presented to the user if and when the condition becomes true. The expression language used in conditions is the same as the expressions used in computed questions. Grouping does not have any

```
form Box1HouseOwning {
   hasSoldHouse: “Did you sell a house in 2010?” boolean
   hasBoughtHouse: “Did you by a house in 2010?” boolean
   hasMaintLoan: “Did you enter a loan for maintenance/reconstruction?”
boolean
   if (hasSoldHouse) {
     sellingPrice: “Price the house was sold for:” money
     privateDebt: “Private debts for the sold house:” money
     valueResidue: “Value residue:” money(sellingPrice - privateDebt)
   }
}
```

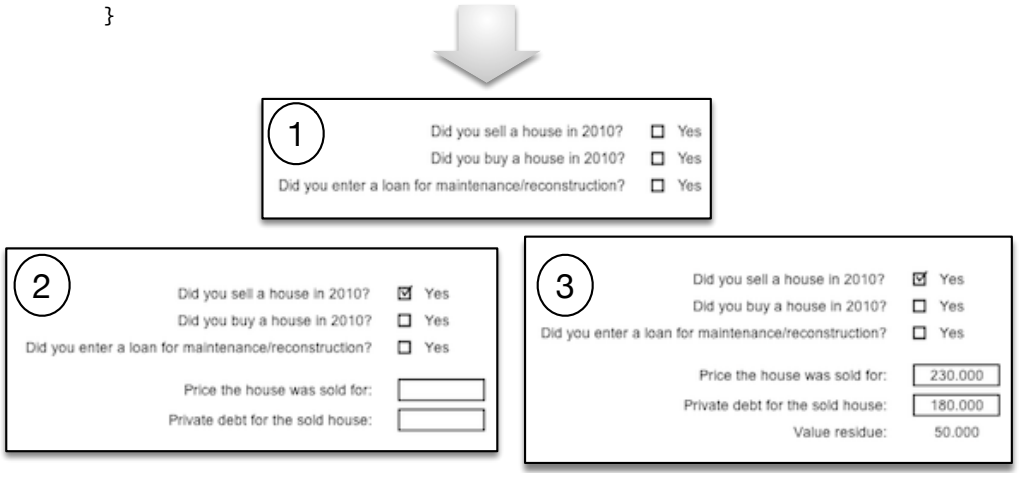


Figure 7.4.: Example of a questionnaire from the “Language Workbench Challenge” 2013

> semantics except to associate a single condition to multiple questions at once."

As for expressions, the original specification describes a number of boolean, comparison, and basic arithmetic expressions that should be supported with their standard semantics. We use a subset of these operations, described below in our ScalaSPL specification of QL.

Semantics of QL

> "The output of a QL description should be a simple GUI program that shows questions as soon as they become enabled. [...] The user should be able to fill in answers, to which the system responds with more questions to be filled in and/or with the display of additional computed results. After all questions have been filled in - the fixed point has been reached - the result of the complete questionnaire should be saved somehow (e.g., as XML, YAML, JSON, etc., or in a database)."

Figure 7.4 visualizes an example for a questionnaire from the original description of the "Language Workbench Challenge".

For the purposes of this dissertation, we focus on modeling a type system for QL and hence keep the modeling of the language's semantics to a minimum. For example, we keep our model of user interaction and of a database for storing the results of questionnaires intentionally simple.

7.2.2. Specification of QL in ScalaSPL

We now describe how we model the informal description of QL from the "Language Workbench Challenge" in ScalaSPL. Notably, we add a type system specification for QL.

Listing 7.8 presents the main datatypes for questionnaires:

```
1  sealed trait Exp extends Expression
2  case class constant(aval: Aval) extends Exp
3  case class qvar(qid: QID) extends Exp
4  case class binop(e1: Exp, op: BinOpT, e2: Exp) extends Exp
5  case class unop(op: UnOpT, e: Exp) extends Exp
6
7  sealed trait Entry extends Expression
8  case class question(qid: QID, l: Label, at: AType) extends Entry
9  case class value(qid: QID, at: AType, exp: Exp) extends Entry
10 case class defquestion(qid: QID, l: Label, at: AType) extends Entry
11 case class ask(qid: QID) extends Entry
12
13 sealed trait Questionnaire extends Expression
14 case class qempty() extends Questionnaire
15 case class qsingle(entry: Entry) extends Questionnaire
16 case class qseq(qs1: Questionnaire, qs2: Questionnaire) extends Questionnaire
```

```
17 case class qcond(e: Exp, thn: Questionnaire, els: Questionnaire) extends Questionnaire
18 case class qgroup(gid: GID, qs: Questionnaire) extends Questionnaire
```

Listing 7.8: QL syntax in ScalaSPL

Top-level expressions in our specification of QL are of type `Questionnaire`. We model the empty questionnaire with `qempty`. A single entry within a questionnaire is of type `Entry`, wrapped in the `Questionnaire` constructor `qsingle`. We model the sequential succession of questionnaires with constructor `qseq` and conditional questions with constructor `qcond`. Constructor `qgroup` allows for hierarchical grouping of questionnaires.

Single questionnaire entries (type `Entry`) may either be

- Questions (constructor `question`), consisting of a question ID, a question label, and a question type (`AType`), the answer type a question expects as the original specification of QL specifies
- "Computed" questions (constructor `value`), which consist of an expression whose value is computed when the question is asked
- "Defined" questions (constructor `defquestion`), which allow for defining a question without displaying it so that it may be defined once and then for example asked in different conditional blocks. The option of defined questions was not explicitly mentioned in the original specification of QL, but is a natural addition that allows for making the type system more interesting.
- "Ask directives" (constructor `ask`) that allow for actually asking questions that have previously been defined via `defquestion`

All entries have a question ID for unique referencing of questions. We leave the type `QID` for question IDs and the type `Label` for question labels undefined in our specification. As answer types, we support Booleans, natural numbers, and "text". We could easily add further types. We implement basic unary and binary operations on these types.

Expressions (type `Exp`) appear as condition within conditional questions as well as in "computed" questions. They may consist of constants (i.e. a value of the three types we support within type `Aval`), of references to the answers of questions via their question IDs, and of binary/unary operations (constructors `binop` and `unop`). We model basic operations on natural numbers (addition, substraction, multiplication, division) as well as basic Boolean and equality operations (i.e. "greater than", "less than", "and", "or", "not" and "equal to").

Listing 7.9 presents our simple model of user interaction within QL:

```
1 def askYesNo(l: Label): YN = ???
2 def askNumber(l: Label): nat = ???
3 def askText(l: Label): string = ???
4
5 def getAnswer(l: Label, at: AType): Aval = (l, at) match {
```

```
6   case (l, YesNo()) => B(askYesNo(l))
7   case (l, Number()) => Num(askNumber(l))
8   case (l, Text()) => T(askText(l))
9 }
10
11 sealed trait AnsMap extends Expression
12 case class aempty() extends AnsMap
13 case class abind(qid: QID, aval: Aval, al: AnsMap) extends AnsMap
```

Listing 7.9: Modeling user interaction with questionnaires in ScalaSPL

We first define three underspecified "oracle" functions for modeling user interaction. From each of these functions we simply assume that given a question label, they produce an answer of a certain type. Function `getAnswer` allows for passing a desired answer type in addition to the question label and makes sure to call the appropriate user oracle function.

For saving the answers to questions, we define a simple list of pairs of question IDs and answer values (type `AnsMap`). Similarly (omitted in the listing above) we define another simple map from question IDs to questions for saving defined questions (called `QMap`). We also define appropriate lookup functions for these two maps.

The top-level reduction function (Listing 7.10) takes a questionnaire, a map with previously given answers (`AnsMap`), and a map with previously defined questions (`QMap`) as arguments and optionally returns an element of type `QConf`, which is a triple that consists of the remaining questionnaire and updated maps for answers and defined questions.

```
1  @Dynamic
2  @ProgressProperty("Progress")
3  @PreservationProperty("Preservation")
4  @Recursive(0)
5  def reduce(q: Questionnaire, ama: AnsMap, qma: QMap): OptQConf = (q, ama, qma)
       match {
6    case (qempty(), _, _) => noQConf()
7    case (qsingle(question(qid, l, t)), am, qm) =>
8      val av = getAnswer(l, t)
9      someQConf(QC(abind(qid, av, am), qm, qempty()))
10   case (qsingle(value(qid, t, exp)), am, qm) =>
11     if (expIsValue(exp))
12       someQConf(QC(abind(qid, getExpValue(exp), am), qm, qempty()))
13     else {
14       val eOpt = reduceExp(exp, am)
15       if (isSomeExp(eOpt))
16         someQConf(QC(am, qm, qsingle(value(qid, t, getExp(eOpt)))))
17       else noQConf()
18     }
19   case (qsingle(defquestion(qid, l, t)), am, qm) =>
20     someQConf(QC(am, qmbind(qid, l, t, qm), qempty()))
21   case (qsingle(ask(qid)), am, qm) =>
22     val qOpt = lookupQMap(qid, qm)
23     if (isSomeQuestion(qOpt))
```

```
24        someQConf(QC(am, qm, qsingle(question(qid,
25          getQuestionLabel(qOpt),
26          getQuestionAType(qOpt)))))
27      else noQConf()
28    case (qseq(qempty(), qs), am, qm) => someQConf(QC(am, qm, qs))
29    case (qseq(qs1, qs2), am, qm) =>
30      val qcOpt = reduce(qs1, am, qm)
31      if (isSomeQC(qcOpt))
32        someQConf(qcappend(getQC(qcOpt), qs2))
33      else noQConf()
34    case (qcond(constant(B(yes())), qs1, qs2), am, qm) => someQConf(QC(am, qm, qs1))
35    case (qcond(constant(B(no())), qs1, qs2), am, qm) => someQConf(QC(am, qm, qs2))
36    case (qcond(e, qs1, qs2), am, qm) =>
37      val eOpt = reduceExp(e, am)
38      if (isSomeExp(eOpt))
39        someQConf(QC(am, qm, qcond(getExp(eOpt), qs1, qs2)))
40      else noQConf()
41    case (qgroup(_, qs), am, qm) => someQConf(QC(am, qm, qs))
42  }
```

Listing 7.10: QL reduction semantics in ScalaSPL (top-level function)

For asking normal questions (case `qsingle(question(qid, l, t))` from line 7 to 9 in Listing 7.10), we call the user answer oracle and then save the resulting answer value in the returned answer map. For computed questions (lines 10 to 18), we reduce first the associated expression via the auxiliary function `reduceExp`. If the expression was already reduced to an expression value, we save this value in the returned answer map. We save a defined question (lines 19 to 20) into the returned map for defined questions. When asking a previously defined question (lines 21 to 27) we look the corresponding question ID up in the map for defined questions and create a normal question out of the entry. We reduce sequential blocks of questions (`qseq`, lines 28 to 33) by first reducing the left part of a block, and then the right one. For conditional questions (lines 34 to 40), we first reduce the guard expression to a boolean value (`B(yes())` or `B(no())`) via the `reduceExp` auxiliary function, and then return either the first, or the second questionnaire within the conditional function. Groups of questions (`qgroup`, line 41) simply reduce to their inner questionnaire.

The reduction of a questionnaire may get stuck at multiple points, e.g. if an expression within a computed or conditional question refers to a question ID that is not present within the given answer map, or if a question is asked (via `ask`) before it was defined (via `defquestion`). We define a type system for QL (Listing 7.11) that syntactically checks such issues before executing a questionnaire. Additionally, our type system ensures that there are no duplicate question IDs. Duplicate question IDs might not directly lead to stuck questionnaires, but may lead to unexpected behavior, so it is desirable to exclude such questionnaires during type-checking.

```
1 sealed trait MapConf extends Context with Type
2 case class MC(atm: ATMap, qtm: ATMap) extends MapConf
```

```
3
4  @Axiom
5  def Tqempty(atm: ATMap, qm: ATMap): Unit = {
6  } ensuring MC(atm, qm) |− qempty() :: MC(atm, qm)
7
8  @Axiom
9  def Tquestion(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType): Unit = {
10   require(lookupATMap(qid, atm) == noAType())
11   } ensuring MC(atm, qm) |− qsingle(question(qid, l, at)) :: MC(atmbind(qid, at, atm), qm)
12
13 @Axiom
14 def Tvalue(qid: QID, atm: ATMap, exp: Exp, qm: ATMap, at: AType): Unit = {
15   require(lookupATMap(qid, atm) == noAType())
16   require(echeck(atm, exp) == someAType(at))
17 } ensuring MC(atm, qm) |− qsingle(value(qid, at, exp)) :: MC(atmbind(qid, at, atm), qm)
18
19 @Axiom
20 def Tdefquestion(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType): Unit = {
21   require(lookupATMap(qid, qm) == noAType())
22 } ensuring MC(atm, qm) |− qsingle(defquestion(qid, l, at)) :: MC(atm, atmbind(qid, at,
       qm))
23
24 @Axiom
25 def Task(qid: QID, qm: ATMap, at: AType, atm: ATMap): Unit = {
26   require(lookupATMap(qid, qm) == someAType(at))
27   require(lookupATMap(qid, atm) == noAType())
28 } ensuring MC(atm, qm) |− qsingle(ask(qid)) :: MC(atmbind(qid, at, atm), qm)
29
30 @Axiom
31 def Tqseq(atm: ATMap, qm: ATMap, q1: Questionnaire, atm1: ATMap,
32         atm2: ATMap, qm1: ATMap, q2: Questionnaire, qm2: ATMap): Unit = {
33   require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
34   require(MC(atm1, qm1) |− q2 :: MC(atm2, qm2))
35 } ensuring MC(atm, qm) |− qseq(q1, q2) :: MC(atm2, qm2)
36
37 @Axiom
38 def Tqcond(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
39         atm1: ATMap, qm1: ATMap, q2: Questionnaire): Unit = {
40   require(echeck(atm, exp) == someAType(YesNo()))
41   require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
42   require(MC(atm, qm) |− q2 :: MC(atm1, qm1))
43 } ensuring MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atm1, qm1)
44
45 @Axiom
46 def Tqgroup(atm: ATMap, qm: ATMap, q: Questionnaire, atm1: ATMap, qm1: ATMap, gid:
       GID): Unit = {
47   require(MC(atm, qm) |− q :: MC(atm1, qm1))
48 } ensuring (MC(atm, qm) |− qgroup(gid, q) :: MC(atm1, qm1))
```

Listing 7.11: QL type system in ScalaSPL

The typing judgment for questionnaires takes a pair of maps from question IDs

to answer types (type `ATMap`) as "input context" and produces a map with possibly updated bindings as "output type". Hence, the type `MapConf` in lines 1 and 2 from Listing 7.11 extends *both* trait `Context` and trait `Type`. The first answer type map saves the IDs and answer types of questions that have been asked previously ("answer type map"), the second map the IDs and answer types of defined questions ("question type map"). Typing rule `Tquestion` first checks that the question ID of the given normal `question` is not present yet in the answer type map, and then simply adds a corresponding binding to the answer type map. Similarly, typing rule `Tdefquestion` checks that the ID of the defined question is not yet present in the question type map and then adds a corresponding entry to that map. Typing rule `Tvalue` also checks that the ID of the computed question is not yet present in the given answer type map, then uses the auxiliary static function `echeck` to type-check the expression of the defined question, ultimately adding a binding from the ID of the question to the type of its expression to the answer type map. Typing rule `Task` checks that 1) the ID of the question to be asked is indeed present in the question type map and that 2) the answer type map does not yet contain a binding for the question ID, i.e. that the question has not been asked yet. Then, the rule adds a binding from the question ID to its question type.

Typing rule `Tseq` type-checks the individual successive questionnaires one after the other, passing the answer type bindings from the first to the second for typing. Ultimately, the type of a `qseq` block consists of all the answer types and question types added by both argument questionnaires. Typing rule `Tqcond` first uses the auxiliary static function `echeck` to check that the expression within the condition of a conditional question has the boolean type `YesNo()`. Then it checks that both argument questionnaires of the conditional question yield the same answer and question type maps, which ultimately become the type of the conditional question. This is a conservative approach for typing, similar to the standard simple typing of the `Ifelse` construct within our running example of typed arithmetic expressions. Finally, typing rule `Tqgoup` simply just requires that its inner questionnaire type-checks successfully.

Finally, we define the progress and preservation theorem for the type system of QL in Listing 7.12. We link the two top-level properties to the top-level reduction function `reduce` of QL via annotations (line 2 and 3 in Listing 7.10).

```
1  @Property
2  def Progress(am: AnsMap, qm: QMap, q: Questionnaire, atm: ATMap,
3        qtm: ATMap, atm2: ATMap, qtm2: ATMap): Unit = {
4    require(!isValue(q))
5    require(typeAM(am) == atm)
6    require(typeQM(qm) == qtm)
7    require(MC(atm, qtm) |- q :: MC(atm2, qtm2))
8  } ensuring exists((am0: AnsMap, qm0: QMap, q0: Questionnaire) =>
9    reduce(q, am, qm) == someQConf(QC(am0, qm0, q0)))
10
11 @Property
12 def Preservation(atm: ATMap, qtm: ATMap, q: Questionnaire, atm1: ATMap, qtm1:
       ATMap, am: AnsMap, qm: QMap, amr: AnsMap, qmr: QMap, qr: Questionnaire, atmr:
```

```
    ATMap, qtmr: ATMap): Unit = {
13  require(MC(atm, qtm) |− q :: MC(atm1, qtm1))
14  require(typeAM(am) == atm)
15  require(typeQM(qm) == qtm)
16  require(reduce(q, am, qm) == someQConf(QC(amr, qmr, qr)))
17  require(atmr == typeAM(amr))
18  require(qtmr == typeQM(qmr))
19 } ensuring (MC(atmr, qtmr) |− qr :: MC(atm1, qtm1))
```

Listing 7.12: Progress and preservation properties for QL's type system in ScalaSPL

In the definition of both properties, we use the two auxiliary functions `typeAM` and `typeQM` for linking the maps used within the dynamic semantics for storing answers and defined questions to the respective answer type maps from the static semantics: The two functions type all values respectively questions to obtain the answer and question type maps referred to in the premises with static conditions. In the preservation theorem, we type the questionnaire `qr`, which is the result of a step of `q`, in the context of the typed maps resulting from the step (`typeAM(amr)` and `typeQM(qmr)`): A step of `q` may update the map of answers or of questions. However, ultimately the maps resulting from typing `q` as well as `qr` have to be identical.

7.2.3. Automated Generation of Proof Graph

Top-level proof steps We first study the proof graph generated for the top-level progress and preservation proof from the ScalaSPL specification for QL with the initial annotations presented in the previous subsection. Without adding more auxiliary lemmas, the initial proof graph consists of two trees, one for each top-level property. Both trees are structurally equal, since they are both generated from the definition of the `reduce` function for QL.

Figure 7.5 visualizes the progress part of this proof graph. We apply Vampire 4.1 in casc mode with a timeout of 150 seconds to all generated proof steps that are not structural induction steps, thus obtaining the intermediate verification state visualized by the node colors in Figure 7.5. As we can see, Vampire 4.1 proves the simple cases `Progressqempty`, `Progressqseq`, and `Progressqgroup` completely. Within the remaining cases (`Progressqsingle` and `Progressqcond`) we also have several proved proof steps, but also three inconclusive sub-cases.

Adding auxiliary lemmas via annotations Next, we refine the inconclusive sub-cases from Figure 7.5 by adding auxiliary progress and preservation properties for the two auxiliary functions called within these cases: `reduceExp` and `lookupQMap`. Since both functions can also fail, we may require progress as well as preservation properties within the top-level progress and preservation proofs. We add these four properties as well as the corresponding annotations to the ScalaSPL specification of QL (see Listing 7.13).

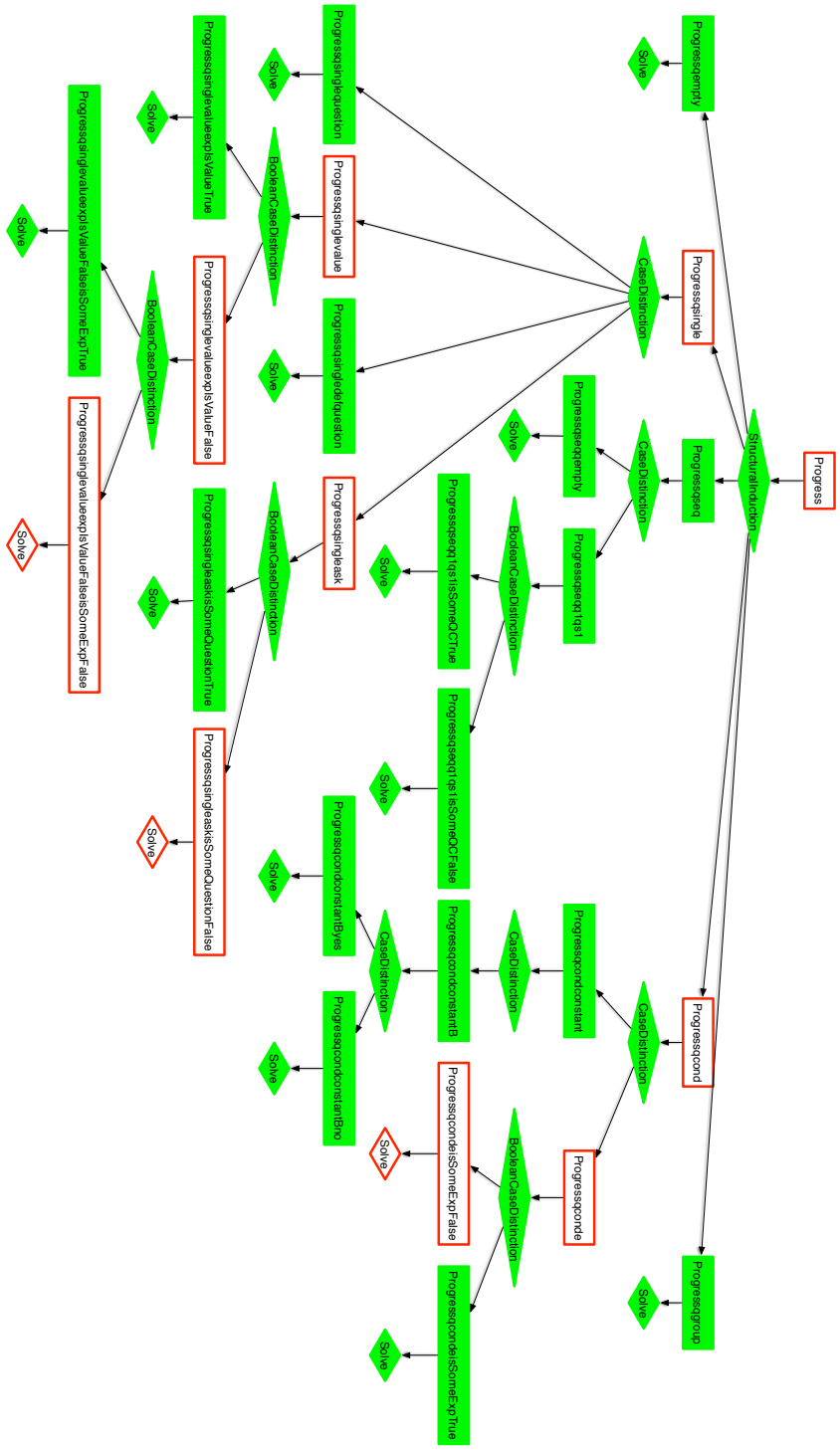


Figure 7.5.: Top-level proof graph for progress proof of QL, with intermediate verification state

```
@Dynamic
@ProgressProperty("reduceExpProgress")
@PreservationProperty("reduceExpPreservation")
@Recursive(0)
def reduceExp(exp: Exp, amap: AnsMap): OptExp = ...

@Dynamic
@ProgressProperty("lookupQMapProgress")
@PreservationProperty("lookupQMapPreservation")
@Recursive(1)
def lookupQMap(id: QID, qm: QMap): OptQuestion = ...

...

@Property
def reduceExpProgress(e: Exp, am: AnsMap, atm: ATMap, at: AType): Unit = {
  require(!expIsValue(e))
  require(typeAM(am) == atm)
  require(echeck(atm, e) == someAType(at))
} ensuring exists((eres: Exp) => reduceExp(e, am) == someExp(eres))

@Property
def reduceExpPreservation(e: Exp, am: AnsMap, atm: ATMap, at: AType, er: Exp): Unit = {
  require(echeck(atm, e) == someAType(at))
  require(typeAM(am) == atm)
  require(reduceExp(e, am) == someExp(er))
} ensuring(echeck(atm, er) == someAType(at))

@Property
def lookupQMapProgress(qm: QMap, qtm: ATMap, qid: QID, at: AType): Unit = {
  require(typeQM(qm) == qtm)
  require(lookupATMap(qid, qtm) == someAType(at))
} ensuring exists((qid0: QID, l0: Label, t0: AType) =>
  lookupQMap(qid, qm) == someQuestion(qid0, l0, t0))

@Property
def lookupQMapPreservation(qm: QMap, atm: ATMap, qid: QID, at: AType, l: Label, t:
    AType): Unit = {
  require(lookupATMap(qid, atm) == someAType(at))
  require(lookupQMap(qid, qm) == someQuestion(qid, l, t))
  require(typeQM(qm) == atm)
} ensuring(at == t)
```

Listing 7.13: Progress and preservation properties for QL's type system in ScalaSPL

This addition refines the generated progress part of the overall proof graph as depicted in Figure 7.6 (where we omitted the branches already shown in Figure 7.5). We apply again Vampire 4.1 in casc mode with a timeout of 150 seconds on the refined proof graph, obtaining the intermediate verification state from Figure 7.6.

The right-hand side of Figure 7.6 shows how the previously inconclusive sub-case of `Progressqsingleask` was refined: The proof strategies added a lemma application

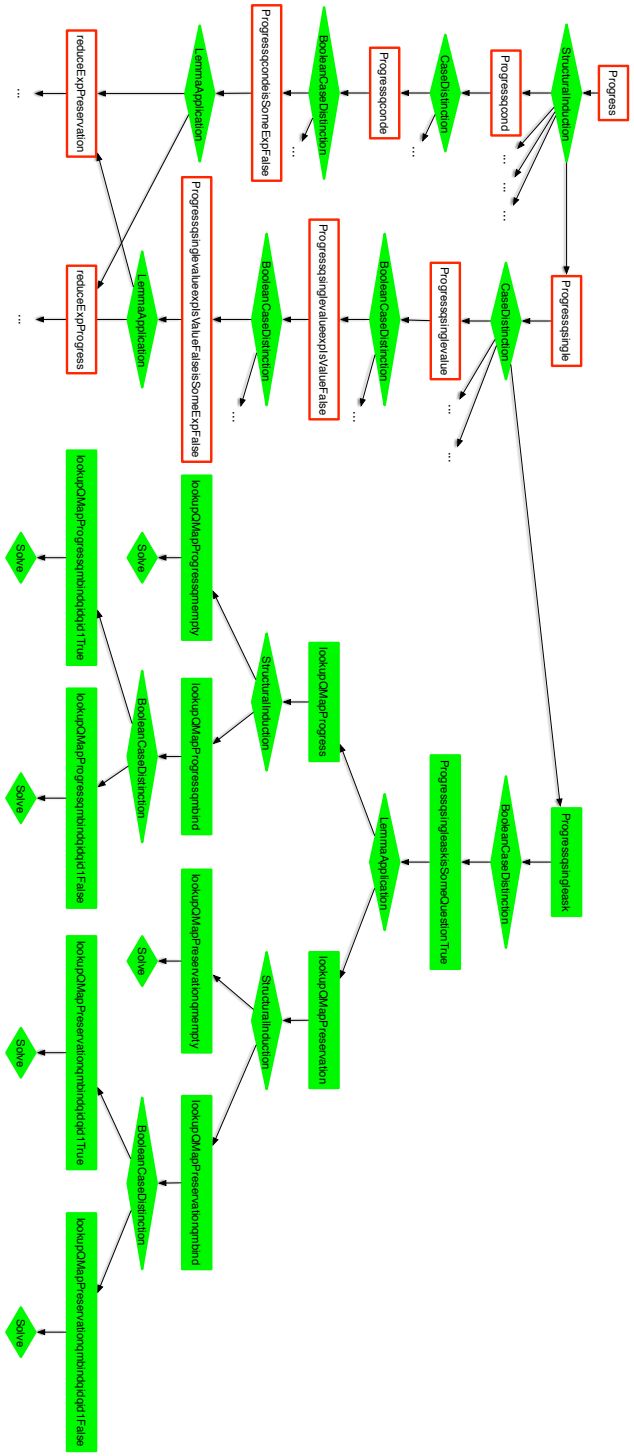


Figure 7.6.: Excerpt of proof graph from progress proof of QL: Refined proof steps for different sub-cases via application of auxiliary lemmas

step with the two auxiliary properties for `lookupQMap` (`lookupQMapProgress` and `lookupQMapPreservation`), along with the corresponding proof steps for these properties. Vampire 4.1 proves all of the generated sub-cases for the auxiliary properties as well as the lemma application step.

On the left-hand side of Figure 7.6, we can see that both the previously inconclusive sub-cases of the induction cases `Progressqcond` and `Progressqsinglevalue` now also obtained lemma application steps with lemmas `reduceExpProgress` and `reduceExpPreservation` and that Vampire 4.1 proves both lemma application steps. The proof strategy also generates proof steps for these two lemmas, which we omitted in Figure 7.6. As the red color of both lemma obligations indicate, there is at least one inconclusive sub-case in each of their proofs, ultimately preventing that we can mark the top-level induction cases on the left-hand side of Figure 7.6 as verified.

We now look more closely at the generated proof steps for `reduceExpProgress`, the relevant lemma for the top-level progress proof. Figure 7.7 visualizes the sub-proof graph for `reduceExpProgress`. We already added additional auxiliary properties for the auxiliary functions that `reduceExp` calls and that these functions in turn call, together with the corresponding function annotations. These additions can be found in the full ScalaSPL specification of QL in Appendix A. In Figure 7.7, we omitted the corresponding generated lemma application steps as well as the generated proof steps for the auxiliary lemmas to keep the graph simpler.

We apply again Vampire 4.1 in casc mode with a timeout of 150 seconds on the refined proof graph. In Figure 7.7, we can see that the sub-cases of `reduceExpProgress` on the right-hand side are completely verified. Vampire 4.1 also verified the lemma application steps within the `qvar` and `unop` cases that we omitted in the figure, as well as the proof steps of these lemmas (which are progress and preservation properties for the functions `lookupAnsMap` and `evalUnOp`). However, there remains a single inconclusive sub-case, namely a sub-case of `reduceExpProgressbinop`: the case where the auxiliary function `evalBinOp` is called. The omitted proof graph part for this case contains another (verified) lemma application step. Some steps within the generated sub-proof graphs for the corresponding lemmas were inconclusive.

7.2.4. Evaluation: Verification of Proof Steps

In Table 7.3, we give an overview of the complete verification results we obtained for the QL case study.

For the fully annotated ScalaSPL specification of QL, our automated proof strategies generate a proof graph with 142 individual proof steps. Out of these steps, 8 steps are top-level structural induction steps for which we cannot use the first-order ATPs and SMT solvers (2 applications for the top-level theorems, 6 for the auxiliary lemmas that we added). For the structural induction steps, we use an internal induction scheme checker, as described above.

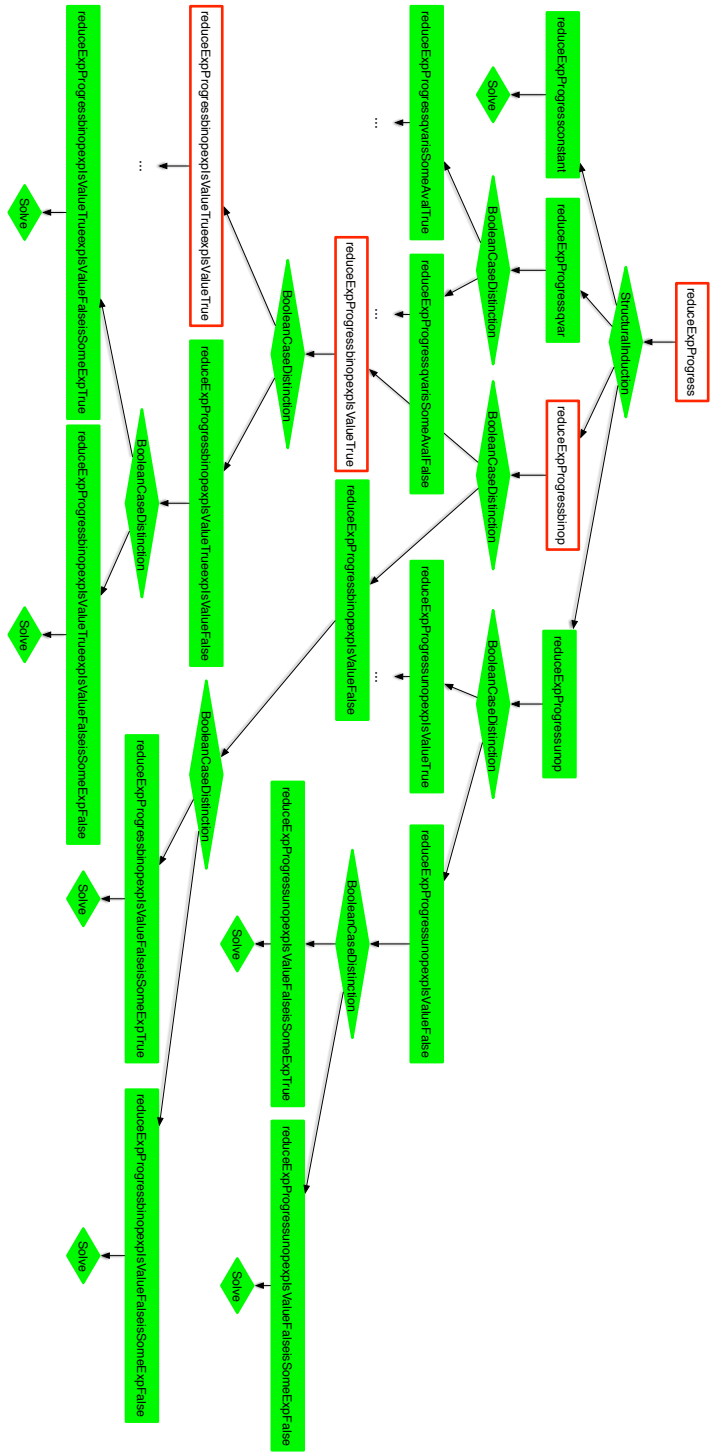


Figure 7.7.: Sub-proof graph from progress proof of QL: Proof steps for auxiliary lemma `reduceExpProgress`

	Total	Proved	Inconclusive
Induction steps	8	8	0
Non-induction steps	134	126	8
Total	**142**	**134**	**8**

Table 7.3.: Overview of proof steps and their verification status within the QL case study

Problem Category	Proved	Inconclusive	%
Proof by definition	65	3	4.6
Case distinctions	42	2	4.7
IH applications	10	1	10
Lemma applications	9	2	22.2

Table 7.4.: Categorization of generated proof problems for QL, together with verification state

Consistency checks Like for the typed SQL case study, we run Vampire 4.1 for 300 seconds on every problem, adding the goal as an axiom and attempting to prove false. All Vampire runs were inconclusive, so we are confident that there are no direct inconsistencies or contradictions within the specifications.

Verification results for induction cases Next, we apply provers to all 134 proof steps in the generated graph that are not structural induction steps. For all steps that are not structural induction steps, we obtain the best results with Vampire 4.1, with a timeout of 150 seconds. We obtain proofs for 126 of these problems within this timeout, i.e. for roughly 89 percent of all problems.

To put the inconclusive problems into perspective, we categorize all generated proof problems that are not structural induction steps like we did for our first case study (see Subsection 7.1.3). Table 7.4 gives an overview of how many problems there are in each category, for how many of these Vampire 4.1 finds proofs, and how many problems in each category remained inconclusive.

We can see that the majority of problems in the QL case study are proofs by definition, followed by case distinctions. Only a smaller part of the problems belong to the more complex categories of applications of induction hypotheses and general lemma applications. We can see that there are a few unsolved problems in every category. Relatively speaking, the highest percentage of unsolved problems remains in the two last, more complex categories.

7.3. Comparison and Discussion

We first present and discuss our observations for the typed SQL and QL case studies. In particular, we summarize and discuss the human effort that was necessary for

obtaining the results of our case studies. Next, we generalize our observations to other DSLs from our target verification domain, motivating the choice of our case studies. Finally, we compare our observations regarding the use of VeriTaS against our observations and conclusions from Chapter 3.

7.3.1. Case Study Comparison: Typed SQL vs. QL

We compare different individual aspects of our two case studies against each other, ranging from the size of the specifications to the verification results we presented.

Specification Both ScalaSPL specifications of our two case studies roughly have the same size. The QL study is a bit smaller than the SQL study if one purely considers lines of code. However, there are some redundancies within the specification of SQL: The specifications of the set queries (`Union`, `Intersection`, and `Difference`) are nearly identical, whereas the different cases within the QL study are more different from each other.

Number of needed auxiliary lemmas In both case studies, we needed to add auxiliary progress and preservation properties to the specification in order to trigger the automated refinement of the initial proof structure generated for the top-level progress and preservation theorems. The typed SQL case study requires more auxiliary lemmas (24 lemmas) than the QL case study (10 lemmas). This is the case since the SQL study uses more auxiliary functions than the QL case study. Additionally, some of the functions of typed SQL require not only a single, but two separate preservation properties (typically one for proving the welltypedness of the function's result, and another one for proving that the function preserves the row count of tables).

Distribution of proof problems The distribution of proof problems between the two case studies differs considerably: While in the typed SQL case study, the majority of proof problems are case distinctions and applications of induction hypotheses, proofs by definitions and also case distinctions dominate the QL study. This is due to the nature of the auxiliary functions within the two studies: In the typed SQL study, almost all auxiliary functions are recursive, i.e. there are a lot of cases that require applying an induction hypotheses. The QL study, on the other hand, contains less recursive functions than the typed SQL study, hence there are many more proof problems that could be proven just by definition. The typed SQL study naturally has more lemma application steps than the QL study since it also requires more auxiliary lemmas (see previous paragraph).

Number and nature of inconclusive problems Both case studies have the same absolute number of inconclusive problems (8). Relatively speaking, the percentage of inconclusive problems is a little higher in the QL case study (ca. 11%) than in the typed SQL study (ca. 5%). Most notably, the QL case study contains a few inconclusive problems in the less complex problem categories of proofs by definition

and case distinctions, whereas the typed SQL case study has no inconclusive problems in these categories. This is probably due to the fact that the QL specification contains a few auxiliary functions with a relatively large number of different top-level individual cases (large compared to the typed SQL case study). For example, the function `evalBinOp` from the QL specification, which evaluates the different supported binary operations on expressions, has 10 top-level cases:

```
1  @Dynamic
2  @ProgressProperty("evalBinOpProgress")
3  @PreservationProperty("evalBinOpPreservation")
4  def evalBinOp(op: BinOpT, av1: Aval, av2: Aval): OptExp = (op, av1, av2) match {
5    case (addop(), Num(n1), Num(n2)) => someExp(constant(Num(plus(n1, n2))))
6    case (subop(), Num(n1), Num(n2)) => someExp(constant(Num(minus(n1, n2))))
7    case (mulop(), Num(n1), Num(n2)) => someExp(constant(Num(multiply(n1, n2))))
8    case (divop(), Num(n1), Num(n2)) => someExp(constant(Num(divide(n1, n2))))
9    case (gtop(), Num(n1), Num(n2)) => someExp(constant(B(gt(n1, n2))))
10   case (ltop(), Num(n1), Num(n2)) => someExp(constant(B(lt(n1, n2))))
11   case (andop(), B(b1), B(b2)) => someExp(constant(B(and(b1, b2))))
12   case (orop(), B(b1), B(b2)) => someExp(constant(B(or(b1, b2))))
13   case (eqop(), a, a1) =>
14     if(a == a1)
15       someExp(constant(B(yes())))
16     else
17       someExp(constant(B(no())))
18   case (_, _, _) => noExp()
19 }
```

Listing 7.14: An auxiliary function within the ScalaSPL specification of QL with a large number of different top-level cases

When generating proof problems for properties about this function, some of the individual sub-cases will receive a large number of generated case premises, notably of premises about "negative" cases. For instance, when generating a proof problem for the last case of `evalBinOp`, the "default case" (line 18 in Listing 7.14), this case will contain 9 generated premises that exclude the previous cases. These additional premises complicate any top-level case distinctions for properties about `evalBinOp` as well as the proofs of the individual sub-cases. And in fact, all inconclusive problems from the category "case distinctions" and two of the problems from the category "proof by definition" arise from generated problems for properties about `evalBinOp`.

The inconclusive problems in the problem categories "IH applications" and "lemma applications" may be explained by the more difficult nature of these problems: These are the steps that would also be more difficult for a human to prove. Still, we were able to break these problems down sufficiently so that ATPs solve a large majority of problems within these two categories. It is not surprising that some smaller percentage of inconclusive problems of this kind remain, given the undecidable nature of the overall verification problem we are tackling in this thesis.

7.3.2. Human effort for case studies

Both our case studies required to add auxiliary progress and preservation properties for all the functions used within the dynamic reduction semantics. This in itself does not require a lot of time from a domain expert who understands the general structure of progress and preservation proofs. For example, the author of this thesis defined all auxiliary properties used within the QL study within less than half a work day, and could then immediately obtain the proof graph with the intermediate verification state sketched in Section 7.2 by applying the automated proof strategies presented in Chapter 6. One could probably develop the auxiliary lemmas needed for the typed SQL study similarly fast (scaling the time to the higher number of lemmas necessary). But note that this is a retro-active claim, since we developed the typed SQL study in two different interactive theorem provers before translating it to ScalaSPL and applying our strategies to it. Hence, we cannot viably assess the necessary time for coming up with the auxiliary properties anymore.

A non-domain expert might have more difficulties in coming up with the necessary auxiliary properties. To help remedy this problem, the parametric and flexible nature of VeriTaS allows us to add lemma generation strategies to the proof strategies we present in this thesis to help with that. We will discuss this further in Chapter 10.

As we have seen in both our case studies, when having all necessary auxiliary lemmas, then one may obtain a proof graph where the majority of problems is verified by an external prover with relatively little effort: It may be necessary to run different provers with different settings or timeouts on the graph, but it is feasible to obtain a good intermediate verification state within a day's time (where trying different provers on inconclusive problems could easily be automated so that no further user interaction is required).

However, we have also seen that inconclusive problems remain, due to the general undecidability of the logic within which the high-level progress and preservation proofs operate (higher-order logic) and due to the incomplete nature of the heuristics used in typical ATPs. These remaining problems will have to be inspected manually. This means users will have to look at the inconclusive proof problems and figure out whether the goals in these problems actually hold or not. For this, proof problems may be pretty-printed into the ScalaSPL format, but may also be displayed in their encoded variants (TPTP or SMTLIB) if necessary. Users may then add further specialized axioms manually to the generated proof graph. Alternatively, users may experiment with changing the specification (for example, reformulating one big function as several smaller ones). Finally, they may attempt to prove the refined proof steps automatically by calling the external provers.

7.3.3. Generalizing to other DSLs

We generalize our observations for typed SQL and QL to other simple DSLs and motivate why this generalization is valid.

Typed SQL and QL as representative simple DSLs Firstly, both our case studies are relevant in practice: SQL is a language that is widely known and used in

practice. QL is widely used within DSL research (via the language workbench competition [Erd+13]). Hence, we cover both language from a practical context and from a research context, which are the two main areas where DSLs are developed and used.

Each of our two case studies is a typical representative of what we call simple DSLs in this thesis: Both SQL and QL do not contain any first-class binders, which allows us to circumvent the "name-binding problem" during verification (see Section 3.1). In general, simple DSLs contain two kinds of language constructs, both of which are represented within our case studies:

1. Abstraction constructs for basic domain-specific operations. Such basic domain-specific operations are for example row selection and column projection within our typed SQL study, as well as collecting the answer of a single question in our QL study. Often, such operations are only technically complex, but their specification does not require complicated programming language constructs. For example, our specification of column projection in typed SQL requires implementing several very technical auxiliary functions that iterate through a table's rows and perform low-level manipulation of lists of rows to construct new tables. All of these functions may be implemented using the basic specification constructs available in ScalaSPL resp. SPL.

2. Domain-specific recursive language constructs. Such recursive language constructs contain again one or more arbitrary terms of the language as arguments. For example, in our typed SQL study, the union query contains two queries as arguments. In our QL study, the construct for conditional questionnaires takes two questionnaires as arguments.

In order to cover a wide range of interesting aspects of simple DSLs, we chose our two case studies so that they are independent from each other and contain very different kinds of language constructs.

Our typed SQL case study contains technical, low-level auxiliary operations (i.e., row selection and column projection). Especially, it combines these operations to one complex operation, i.e., in the `selectFromWhere` case. This combination of operations firstly leads to a more complex specification of the associated type system- i.e., the typing rule for `selectFromWhere` queries requires more premises and specifications of auxiliary functions on the static side. This more complicated typing rule in turn leads to more complex proof steps within the associated progress and preservation proofs. We have shown that VeriTaS is able to generate such proof steps in a way so that they can be proven automatically by existing ATPs.

Additionally, our typed SQL case study contains mostly recursive specifications of auxiliary functions. Hence, the generated progress and preservation proofs will contain a high number of inductive steps, where the corresponding induction hypotheses have to be applied.

In contrast, our QL case study models different features such as user interaction and the generation of collections for questionnaire answers. Our specifications of QL contains mostly basic operations (such as numeric addition), but many of these.

Consequentially, we have obtain auxiliary functions within the specification that need to cover a large number of cases (such as `evalBinOp`) that we presented above). Furthermore, our QL case study contains less recursive specifications that our typed SQL case study. In combination, these two observations lead to more case distinction steps in the QL case study than in the typed SQL case study, while the number of inductive proof steps is lower.

Hence, our two case studies cover a sufficiently wide range of different language and type system features of other simple DSLs so that we may draw general conclusions on VeriTaS' degree of automation for our target verification domain.

General observations for type soundness proofs of simple DSLs with VeriTaS We summarize our general observations regarding the automation of our target verification domain in VeriTaS. We draw conclusions regarding the overall usability of our instantiation of VeriTaS and its degree of automation.

- *Specification:* Specifying languages and their type systems in our instantiation of VeriTaS is straightforward, using ScalaSPL. Since ScalaSPL is a subset of Scala, we receive all kinds of helpful IDE-support while specifying type systems, such as on-the-fly type-checking of our specifications and auto-completion of names of constructors (case classes) and definitions (methods). Slightly disadvantageous from a user perspective is that we cannot use type parameters for specifying data structures and functions that would allow us to use a data structure or function with multiple types. Furthermore, we cannot use standard library constructs such a lists. However, this is a purely technical limitation of VeriTaS, not a conceptual one. Especially for small DSLs, the additional specification effort is low for end users.

- *Human-readable proofs/verification status:* VeriTaS' visualization of proof structures via proof graphs, along with their verification status, enables to quickly assess the overall proof structure of a progress and preservation proof and to identify inconclusive steps. Users can easily inspect proof steps by pretty-printing them to ScalaSPL.

- *User interaction:* Currently, end users of VeriTaS interact with the verification infrastructure via its API, using their Scala IDE. VeriTaS' API methods allow for easily triggering the generation of proof graphs of progress and preservation proofs as well as the visualization of the generated graphs. The API also allows for inspecting and refining proof graphs manually. Thus, VeriTaS' user interface offers all the necessary elements that we also defined as requirements on an infrastructure for domain-specific verification. The user interface could be improved further, e.g., by adding a GUI. However, this would require engineering effort that was not in the scope of this thesis.

- *Degree of automation:* For specifications of type systems that include all necessary axioms for typing inversion and all necessary progress and preservation properties of auxiliary functions, we estimate that our instantiation of VeriTaS

will typically reach a degree of automation between 80 and 90 %. That is, our proof strategies fully automatically generate proof graphs where 80 to 90 % of the proof steps can be verified automatically by currently existing ATPs, together with a verifier that checks the correctness of the applied induction schemes. The remaining proof steps (10 to 20 %) will consist firstly of some steps that could, in principle, be verified by existing ATPs, but the verification is still inconclusive due to the heuristics that the provers use. For these steps, one may need to try different heuristics and/or different encoding strategies (Chapter 8 contains further information about encoding strategies). For the remaining steps, end users will have to refine the generated proof graph by

1. specifying auxiliary properties that do not directly fit into the scheme of progress and preservation properties that we outlined in Section 6.4 and trigger the generation of a new proof graph that will then also contain lemma application steps with these auxiliary properties and/or
2. manually adding proof steps and sub-obligations to the generated proof graph with intermediate obligations.

7.3.4. VeriTaS vs. Isabelle/HOL

We compare conducting progress and preservation proofs in VeriTaS to conducting such proofs in Isabelle/HOL. We discussed the syntax and features of Isabelle/HOL in detail in Section 3.2.

The specification of type systems within Isabelle/HOL is arguably a bit more comfortable than in our instance of VeriTaS, since ScalaSPL does not support parametric polymorphism and also does not offer existing data structures at this point. This means, as we have seen, that users need to specify specialized list and option constructs etc. themselves within ScalaSPL. However, as we also emphasized above, this limitation of ScalaSPL is of a purely technical nature. Adding a library of data structures and adding type parameters would be conceptually straightforward, but require some engineering effort (in particular, the translation from ScalaSPL to SPL would have to automatically translate such additional language elements to SPL). For smaller specifications such as the ones we presented in this chapter, the additional effort of manually encoding standard data structures is however negligible.

Obtaining progress and preservation proofs for simple DSLs in VeriTaS requires by far fewer user interactions than within Isabelle/HOL, where almost all case distinctions and applications of auxiliary lemmas need to be explicitly spelled out. In contrast, in VeriTaS we only have to provide all necessary auxiliary properties and add at most a couple of domain-specific annotations per function specification. At the same time, a generated and verified proof graph in VeriTaS still offers similar amenities as a proof in Isabelle/HOL: A proof graph with high-level steps offers a human-readable format of a machine-checked proof, just like an Isar script in Isabelle. Furthermore, since the proof steps within a verified proof graph have to be verified by external, trusted provers, a verified proof graph is reliable.

For our case study of typed SQL, we are able to provide exact numbers to compare VeriTaS and Isabelle/HOL, since we conducted the corresponding progress and preservation proof entirely in both systems: In Section 3.2, we reported that conducting these proofs together (excluding the specification of the language and the type system) requires over 1000 lines of Isabelle and Isar code. At least half of the Isar proof commands had to be given manually, the other half, i.e. on average every second Isar command, was generated by Sledgehammer or by Isabelle's system for suggesting high-level proof structures). In VeriTaS, we only have to add 190 lines of auxiliary properties (including the top-level theorems) as well as about 70 lines of code for domain-specific function annotations. For adding these lines in VeriTaS, one does not need to learn or know any specific proof language, but only the ScalaSPL syntax for specifying properties as well as a handful of domain-specific annotations.

7.3.5. VeriTaS vs. Dafny

We compare conducting progress and preservation proofs in VeriTaS to conducting such proofs in Dafny. We discussed the syntax and features of Dafny in detail in Section 3.3.

Just like Isabelle, Dafny currently offers more useful language features for specification than ScalaSPL in VeriTaS. However, VeriTaS offers syntactic sugar for the specifications of typing judgments that Dafny does not. So some parts of a specification may be more effort within ScalaSPL, but type system specifications may be more intuitively readable for domain experts.

Obtaining progress and preservation proofs in Dafny always requires formulating the necessary auxiliary lemmas, as well as proof commands that indicate exactly in which sub-cases which instance of which lemma as to be used. In VeriTaS, lemmas are automatically added at the points where they are needed by the domain-specific strategies. Also, Dafny occasionally requires that the top-level proof structure of a lemma is spelled out. Our proof strategies within VeriTaS mostly generate a correct high-level proof structure automatically (but may also sometimes fail in the general case).

One big advantage of using VeriTaS over Dafny is that VeriTaS generates human-readable proofs in the shape of a proof graph. In Dafny, we cannot directly inspect the parts of proofs that Dafny generates automatically. Especially, our proof strategies in VeriTaS will not simply fail, but generate incomplete or wrong proof graphs, where some parts may be complete and correct. Users my inspect these graphs as well and refine the incorrect steps. If Dafny cannot automatically generate a proof, it just produces a generic top-level error message, as we have seen in Subsection 3.3.3. It is then up to a user to locate the part of the proof where something went wrong.

Again, we can give concrete numbers for comparing our typed SQL case study in VeriTaS to the same case study in Dafny, which we presented in detail in Section 3.3: The files containing the progress and preservation proof for typed SQL in Dafny (including the specifications of auxiliary lemmas) contain together about 370 lines of code, none of which are generated automatically (gaps in the proof script indicate

that Dafny completed the proof automatically in the background for us). The lines representing proof commands require end users again to learn and understand a proof language.

7.4. Summary

We presented two case studies for evaluating the effectivity and efficiency of our automated proof strategies for generating proof graphs for progress and preservation proofs within VeriTaS: the subset of typed SQL we already used for our survey in Chapter 3 and a new case study, the questionnaire language QL. We have seen that for both case studies, our strategies generate sensible proof graphs where a large majority of the generated proof problems can be verified by external ATPs. The more auxiliary progress and preservation properties a user adds to a specification and links to the auxiliary functions via user annotations, the more refined the proof graphs that the automated proof strategies generate get. Ultimately, a small number of inconclusive proof problems remain. We described the nature of the inconclusive problems and suggested how they may be refined manually.

We discussed how and why the results from our two case studies generalize to other simple DSLs and compared VeriTaS to the verification systems we presented in Chapter 3 (Isabelle/HOL and Dafny). We concluded that VeriTaS significantly raises the degree of automation for type soundness proofs of simple DSLs compared to both systems.

Chapter

8

Empirical Study of Encoding Strategies

In Section 5.3, we described how one can connect existing ATPs and SMT solvers to the VeriTaS verification infrastructure. An essential part of connecting verifiers to VeriTaS is encoding proof problems from an input format used in VeriTaS into formats which may be processed by existing provers. For example, for encoding SPL to TPTP, we described a single encoding strategy in Section 5.4. However, there are many alternative encoding strategies. Already our early experiments with encoding proof problems for ATPs revealed that the choice of the encoding strategy may influence the success rates of the ATPs dramatically. In order to find encoding strategies that work well for proving our proof problems and to study this phenomenon in a more systematic way, we conducted an empirical comparison study of encoding strategies.

This chapter describes the study setup, notably including the encoding variants we study (Section 8.1) and test problems that we used (Section 8.2). In Section 8.3, we describe the results of our comparison of encoding strategies.

We also studied an issue that is orthogonal to the encoding strategy: the selection of axioms passed on to an ATP. The overall size of a problem may also influence the heuristics used for proving a certain problem. Since not all axioms are necessarily relevant in every case, one may ask whether problem-specific pre-selection of relevant axiom makes sense. We describe an extension to our empirical study of encoding strategies to answer this question and its results in Section 8.4.

The results we obtained for this study helped us to develop VeriTaS as it is presented in this thesis and to obtain the results for the case studies we presented in Chapter 7.

Remark 8.1. The author of this thesis co-published the content from this chapter in a paper within the journal "Science of Computer Programming" in 2018, under the title "Exploration of language specifications by compilation to first-order

logic" [Gre+18a]. An earlier version of this journal paper appeared in the conference "Principles and Practice of Declarative Programming (PPDP)" in 2016, under the same title [Gre+16]. In this thesis, the content from the journal paper was updated and adapted to the terminology and examples used within the thesis.

Furthermore, the author of this thesis co-published individual smaller comparison studies which are not directly treated in the present chapter on different Vampire workshops [GEM15; GEM16; GPM18]. ◊

8.1. Encoding Alternatives

There are many alternative ways to encode an SPL specification in first-order logic. Our initial experiments with using ATPs on compiled SPL specifications revealed that small differences in the encoding strategy can vastly influence whether a prover can find a proof within a given timeout or reports that a search was inconclusive.

In this section, we describe alternative encoding strategies to the strategy we presented in Section 5.4. Based on our initial experiment, for each variation, we hypothesize why and how it can influence prover performance. A systematic empirical comparison of all variants follows in Section 8.2 (setup of an empirical study) and 8.3 (results of the empirical study).

8.1.1. Encoding Syntactic Sorts

The first dimension for generating alternative encoding strategies concerns the treatment of syntactic sorts like Term and Ty from our running example of typed arithmetic expressions. How should we represent such sorts in first-order logic and how should we declare function symbols that operate on syntactic sorts?

Typed logic In Section 5.4, we used typed first-order logic and represented sorts as types of that logic. We added typed signatures for declarations of function symbols and used types in quantifiers. The advantage of this encoding is that the theorem provers can direclty exploit typing information. However, not all automated theorem provers support typed logics.

Type guards As alternative to a typed logic, one can use untyped logic and encode sorts via type guards, as for example described in [Bla+13b]. Type guards are predicates of the form $\mathsf{guard}_{\mathsf{T}}(\mathsf{t})$ that yield true if and only if term t has sort T. In the encoding described previously in Section 5.4, we declared function symbols for functions, constructors, and constants. Now, instead of each function declaration of the form $\mathsf{f} : \overline{\mathsf{T}} \to \mathsf{U}$, we introduce a guard axiom that describes well-typed usages of f:

$$\forall \mathsf{x}_1, ..., \mathsf{x}_n.\ \mathsf{guard}_{\mathsf{T}_1}(\mathsf{x}_1) \wedge ... \wedge \mathsf{guard}_{\mathsf{T}_n}(\mathsf{x}_n) \Longleftrightarrow \mathsf{guard}_{\mathsf{U}}(\mathsf{f}(\mathsf{x}_1, ..., \mathsf{x}_n))$$

Listing 8.1: Guard axioms for functions when using type guards

Since we encore data type constructors from SPL as functions in FOL, the declarations of the corresponding functions are also replaced by corresponding guard axioms.

For the rest of the specification, we introduce guard calls for all (then untyped) quantified variables as a post-processing step. That is, after data types and functions have been translated into formulas, we apply the following rewritings:

```
1 ∀x : T. φ   ⇝   ∀x. guard_T(x) ⟹ φ
2 ∃x : T. φ   ⇝   ∃x. guard_T(x) ∧ φ
```

Listing 8.2: Rewritings of quantified formulas with type guards

After having applied these rewritings to all quantified formulas, we replaced all types from an encoded SPL specification by type guards. Accordingly, we can pass the resulting encoded SPL specification to any theorem prover that supports untyped first-order logic.

Type erasure While type guards make the encoding amenable to many theorem provers, type guards also increase the number and size of axioms. This may slow down proof search considerably. As an alternative strategy, we can erase typing information from the encoding altogether, with some prior considerations.

In general, the erasure of typing information is unsound, that is, it does not preserve satisfiability [Bla+13b]. In a logic with equality and for sorts with finite domains, type erasure can lead to problems. For example, for singleton sort Unit, formula $(\forall x : \mathsf{Unit}, y : \mathsf{Unit}.\ x = y)$ holds whereas its erasure $(\forall x, y.x = y)$ does not hold in general. This problem occurs whenever a formula is *non-monotonic*, which means the formula puts constraints on the cardinality of a sort's domain. However, type erasure is sound for sorts with infinite domain [CLS11].

In our case, we generate all sorts in encoded proof problems from SPL specification, where we clearly distinguish between *closed* and *open* ADTs (see Section 5.1.1). Thus, we can easily distinguish between sorts with infinite and finite domains: An SPL data type A has an infinite domain if one of the following conditions holds:

1. A is an open data type, which are countably infinite by definition.
2. A is recursive
3. A refers in its definition to another data type that has an infinite domain via at least one argument of a data type constructor.

If none of these three conditions holds, a data type has a finite domain. Since data types are defined via finite lists of data type constructors, we can enumerate all shapes terms of data types with finite domains can take. Thus, we can fully erase all typing information as a post-processing step of the translation described in Section 5.4, which operates as follows:

The left two rewritings above eliminate the typing information for variables of a type with finite domain by inlining the necessary domain information. The right two rewritings erase types with infinite domains from quantifications. After this

```
1 if T = c1(T1) | ... | cn(Tn) has a finite domain:
2   ∀x : T. φ   ⇝   ∀x. (⋁i ∃yi. x = ci(yi)) ⟹ φ
3   ∃x : T. φ   ⇝   ∃x. (⋁i ∃yi. x = ci(yi)) ∧ φ
```

```
1 if T has an infinite domain:
2   ∀x : T. φ   ⇝   ∀x. φ
3   ∃x : T. φ   ⇝   ∃x. φ
```

Listing 8.3.: Type erasure for types with finite and infinite domain

erasure, the domain axioms from Section 5.4.1, point 3 become obsolete, so we drop them when erasing the types of quantified variables.

Like the strategy for sort encoding that uses type guards, type erasure yields compiled SPL specifications which can be used with any first-order theorem prover. But unlike the strategy with type guards, type erasure does not add axioms, and does not increase the size of axioms that quantify over sorts of infinite domains. However, the type-erasure strategy leads to larger axioms if quantification over sorts with finite domains occurs.

8.1.2. Encoding of Bound Variables

The second encoding variation we consider concerns the encoding of bound variables $x = t$, where x is a variable bound by the equation to a term t. Such bindings can occur in user-defined inference rules or result from our transformations of function equations described in Subsection 5.4.2. Is it advisable to retain such equations or should we eliminate them through inlining? Or should we rather do the contrary and introduce such bindings for all subterms of a formula?

Internally, ATPs heuristically apply variable elimination as well as subformula naming strategies [RSV16; AW13], both of which are supposed to generate an optimal internal representation of a given problem. However, even despite this fact, we observed in our initial experiments that already the initial encoding of such bindings can have a huge impact on the performance of provers. This indicates that the decision how to encode bound variables matters already on the user level, and not only within the internals of ATPs.

Unchanged In Section 5.4, we did not specifically consider bound variables and left them unchanged. That is, we reproduced bindings exactly as they occurred in the SPL language specification and exactly how they were generated during our transformations. Our initial encoding strategy from Section 5.4 only introduces variable bindings for *let*-bindings and for function pattern variables $\overline{pv}$ in inversion axioms. Moreover, the type-erasure strategy we presented in Subsection 8.1.1 introduces variable bindings for variables that have a sort with finite domain.

Inlining We can use inlining to eliminate bound variables.[1] This may be beneficial for proof search because the elimination of variables potentially creates more ground terms, which results in fewer inferences.

[1] This process is also called *Equality Resolution* in the literature.

The inlining and elimination of a bound variable $\mathsf{x} = \mathsf{t}$ in a formula ϕ is sound if $\phi \equiv (\mathsf{x} = \mathsf{t}) \Longrightarrow \psi$. We can then replace ϕ by $\psi[\mathsf{x} := \mathsf{t}]$, which eliminates the bound variable x. In our implementation, we conservatively approximate this applicability condition by supporting inlining only for implications that syntactically appear in ϕ. This condition covers all inlining opportunities that occur in our case studies.

For example, in the axiomatized reduce function from Section 5.4.2, inlining eliminates the bound variable $\mathsf{ot_1} = \mathsf{reduce}(\mathsf{t_1})$ in the third axiom (line 5 in Listing 5.3) by inlining $\mathsf{reduce}(\mathsf{t_1})$ in two places within the axiom as follows:

```
1 ∀ t1:Term, t2:Term, t3:Term. isSomeTerm(reduce(t1)) ∧
2   ifelse(t1, t2, t3) ≠ ifelse(true, t2, t3) ∧ ifelse(t1, t2, t3) ≠ ifelse(false, t2, t3)
3     ⟹ reduce(ifelse(t1, t2, t3)) = someTerm(ifelse(getTerm(reduce(t1)), t2, t3))
```

Listing 8.4: Variable inlining for third axiom of reduce function

Variable introduction While inlining reduces the number of variables and literals in a formula, it increases the size of the remaining literals. In particular, when subformulas occur multiple times, instead of inlining, it may be beneficial to introduce new variables and bind them to the recurring subformulas, replacing the repeating occurrences with the newly introduced variable. This reduces the size of the individual literals by increasing the number of literals and variables.

The variable-introduction strategy introduces fresh variable names and bindings for all subformulas, similar to static single assignment. We make sure to reuse the same name for syntactically equivalent subformulas, such that reoccurring subformulas are bound by the same variable. For example, this encoding names all subformulas within the third axiom of the encoding of function reduce (line 5 in Listing 5.3) as follows:

```
1 ∀ t1:Term, t2:Term, t3:Term. ∀ ot1:OptTerm. ∀ t4: Term, t5: Term, t6: Term, t7: OptTerm,
2   t8: Term, t9: Term, t10: OptTerm.
3     isSomeTerm(ot1) ∧ ot1 = reduce(t1) ∧ t4 = ifelse(t1, t2, t3) ∧ t5 = ifelse(true, t2, t3) ∧
4     t6 = ifelse(false, t2, t3) ∧ t7 = reduce(t4) ∧ t8 = getTerm(ot1) ∧
5     t9 = ifelse(t8, t2, t3) ∧ t10 = someTerm(t9) ∧ t4 ≠ t5 ∧ t4 ≠ t6
6       ⟹ t7 = t10
```

Listing 8.5: Example of full subformula naming for third axiom of reduce function

Parameters and result variables Inlining and variable introduction represent two extremes of variable handling. There are several compromises between these two extremes. We tried several alternatives, including common subformula elimination, and ultimately chose to include the strategy that seemed to have the largest effect on our example specifications (see Section 8.2.1) into our study: The strategy leaves variable bindings from the specification unchanged and introduces variable bindings for function parameters and results that appear in conclusions of implications. For example, applying this transformation to the third axiom of function reduce yields:

```
1 ∀ t1:Term, t2:Term, t3:Term. ∀ ot1:OptTerm. ∀ arg: Term, result: OptTerm.
2   arg = ifelse(t1, t2, t3) ∧ result = someTerm(ifelse(getTerm(ot1), t2, t3)) ∧
3   isSomeTerm(ot1) ∧ ot1 = reduce(t1) ∧
4   arg ≠ ifelse(true, t2, t3) ∧ arg ≠ ifelse(false, t2, t3)
5       ⟹ reduce(arg) = result
```

Listing 8.6: Example of naming of parameter and result variables of third axiom of reduce function

8.1.3. Simplifications

The third variation of our encoding concerns logical simplifications. Just like for the encoding of variables, theorem provers also internally conduct general-purpose simplifications. Again, we observed during our initial experiments that in some cases, applying logical simplifications before passing the problems to a first-order theorem prover affected prover performance. So we also study the effects of simplification systematically.

No simplification In Section 5.4, our encoding did not apply any simplifications. Consequently, the resulting formulas may be unnecessarily large. Without further simplification in the encoding, we rely on the preprocessing of the theorem provers.

General-purpose simplifications This encoding exhaustively performs basic general-purpose simplifications like the following ones on all formulas ($\mathsf{fv}(\phi)$ denotes the set of free variables in ϕ):

```
1 x = x ⇝ true
2 true ∧ φ ⇝ φ
3 false ∧ φ ⇝ false
4 φ ∧ φ ⇝ φ
5 false ∨ φ ⇝ φ
```

```
1 true ∨ φ ⇝ true
2 φ ∨ φ ⇝ φ
3 ∀ x̄. φ ⇝ ∀ (x̄ ∩ fv(φ)). φ
4 ∃ x̄. φ ⇝ ∃ (x̄ ∩ fv(φ)). φ
5 ...
```

Domain-specific simplifications We can use domain-specific knowledge about a language's SPL specification to simplify the generated formulas. Since theorem provers are unaware of the original specification, they cannot directly apply such simplifications. Instead, in order to achieve a domain-specific simplification, the provers normally have to do non-local reasoning which takes the entire specification into account. This non-local reasoning may not always be successful (depending on the used heuristics), so that some possible domain-specific simplifications may be applied and some not.

For this study, we focus on investigating domain-specific simplifications for algebraic data types. We apply the following simplifications for equations (and analogously for inequalities) over constructors, where c, c_1, and c_2 are constructor names:

```
1 c(a1,...,an) = c(b1,...,bn) ⇝ a1 = b1 ∧ ... ∧ an = bn
2 c1(a1,...,am) = c2(b1,...,bn) ⇝ false if c1 ≠ c2
```

Listing 8.7: Domain-specific rewritings for constructor equalities/inequalities

These rewritings are justified by the axiomatization we give in Section 5.4.1 for algebraic data types. A theorem prover can do such rewritings itself, but it needs non-local reasoning to find and apply the appropriate axioms for datatypes. Our domain-specific simplification can in particular reduce the size of formulas that encode the pattern matching of functions. For example, our simplification yields the following axioms for the third equation of function reduce, eliminating the inequalities that NPC generates (see Section 5.4.2):

```
1 ∀ t1:Term, t2:Term, t3:Term. ∀ ot1:OptTerm.
2   isSomeTerm(ot1) ∧ ot1 = reduce(t1) ∧ t1 ≠ true ∧ t1 ≠ false
3       ⟹ reduce(ifelse(t1, t2, t3)) = someTerm(ifelse(getTerm(ot1), t2, t3))
```

Listing 8.8: Example for domain-specific simplification in third axiom of reduce function

Remark 8.2. Note that rewritings of this kind for algebraic datatypes are now integrated in a newer version of the Vampire theorem prover [KRV17], which was developed in parallel with our work and published after our original publication of these results [Gre+16]. However, older Vampire versions and other ATPs do not directly support such rewritings internally, so it is still worthwhile to include them in our encoding comparison study. In a recent workshop paper [GPM18], we explored the application of Vampire's new direct support for algebraic datatypes to our specific input problems from the domain of type soundness proofs. There, we did not observe better prover performance than with older Vampire versions. However, this may be due to several different factors, since the heuristics internally used in Vampire keep changing with the Vampire versions. ◊

8.1.4. A Modular and Reusable Compiler Product Line

We presented alternative compilation strategies along three dimensions: 3 alternatives for encoding syntactic sorts, 4 alternatives for handling variables, and 3 alternatives for simplification. Since the three dimensions are independent, this amounts to $3 * 4 * 3 = 36$ different encoding strategies (or *compilation strategies*).

We implemented all of these compilation strategies in a modular and reusable compiler product line within VeriTaS. Our compiler takes an SPL specification as input and first iteratively translates higher-level SPL specification constructs into lower-level SPL constructs. For example, we translate function equations to SPL inference rules. Each single encoding step and notably each different encoding alternative is implemented as a separate transformation. For example, there are different individual transformation steps for adding the axiomatic encoding of data types (see Subsection 5.4.1) or for adding function inversion axioms (see Subsection 5.4.2). This way, users can easily assemble each different compilation

strategy by chaining the appropriate individual transformation steps. It is also possible to easily modify existing transformation steps by inheriting from a specific transformation traits in the code and overriding some of its definitions or to add custom new transformation steps for SPL constructs. One can then integrate new or modified transformation steps into a transformation chain to produce a new compilation strategy. We demonstrated the reusability of our compiler product line in Section 5.5, where we presented the encoding from SPL to SMTLIB: For this encoding, we reused large parts of our compiler product line for encoding proof problems to TPTP.

Eventually, each of our predefined 36 compilation pipelines produces an SPL specification which only contains low-level specification constructs: inference rules (the ones generated from datatype declarations and function equations marked as axioms, the ones from goals marked as goals) without **let** or **if** constructs and function signatures. As a last step, our compiler translates these low-level SPL specifications into the standardized TPTP format [Sut10] that is used in theorem-prover contests and supported by a great number of automated first-order theorem provers. By default, our compiler translates the specification using each of the 36 different compilation strategies in turn. However, the compiler can also accept a description of the desired configuration space, such that it only applies a subset of the available compilation strategies.

8.2. A Comparison Study of Encoding Alternatives

To investigate the effect of different compilation strategies on prover performance, we designed an empirical study using our example specifications of typed SQL and of typed QL (see Chapter 7) For our study, we defined 10 proof goals in each of 5 goal categories (execution, synthesis, testing, verification, counterexample) on both the SQL and the QL specifications (that is, 50 proof goals in total per specification). We describe the categories and the goals in Subsection 8.2.1. Our study aims to answer the following research questions:

RQ1 Do different but equivalent compilation strategies affect prover performance?

RQ2 How does the strategy for encoding syntactic sorts influence prover performance?

RQ3 How does the strategy for encoding variables influence prover performance?

RQ4 How do simplifications influence prover performance?

RQ5 When does domain-specific simplification have an influence on prover performance?

RQ6 Is there a compilation strategy that performs best for all goal categories? Otherwise, what is the best compilation strategy for each goal category?

Based on our initial experiments with different compilation strategies, we expect for RQ1 that our data will confirm that different but semantically equivalent compilation strategies do affect prover performance. For RQ2, RQ3, and RQ4, we expect to observe differences between the different strategies we are investigating and tendencies that indicate which compilation strategies might be better, and which to

avoid. For RQ5, we expect to observe that for shorter timeouts, domain-specific simplification improves prover performance. For RQ6, we expect that there will be certain combinations of strategies that perform best in at least one goal category.

8.2.1. Study Goals on Example Language Specifications

As example language specifications for investigating the effect of different compilation strategies on prover performance, we used the SPL specifications of our two case studies from Chapter 7, typed SQL and QL.

In our study, we distinguish 5 goal categories that explore a language specification in different ways. Below, we describe each of the 5 categories and present example goals. The example goals in this section are all taken from the specification of typed SQL. The goals we defined for the specification of QL are similar.

Execution The first category describes goals that execute part of the language specification on some input in order to retrieve the result of the execution. In principle, using ATPs for this goal category permits the inspection of semantics that are not directly executable, such as indeterministic and denotational semantics. We do not exploit this possibility in our case study, since we focus on the comparison of compilation strategies in this paper.

For executing a function f on some input t, we encode an execution goal in first-order logic as follows:

$$\exists\, \mathsf{v}.\ \mathit{ground}(\mathsf{v}) \wedge \mathsf{f}(\mathsf{t}) = \mathsf{v}$$

That is, we ask whether there is some value v such that f(t) computes v. Since mathematical functions are total and always produce a result, an obvious candidate for v would be f(t) itself. If f(t) is undefined in the original SPL specification, this answer does not yield any insight into the language specification. Therefore, we require that the result of f(t) is equivalent to a ground term: A term satisfies predicate *ground* if it solely consists of calls to data-type constructors and references to constants. This way, we force the ATP to always inspect the axioms that define f.

For our study, we defined 10 execution goals that probe different parts of the dynamic semantics of SQL. Representatively, we show one goal here that explores the auxiliary function rawUnion:

```
1 local {
2   different consts r1, r2, r3, r4 : Row
3           goal
4           t1 == tcons(r1, tcons(r2, tcons(r4, tempty)))
5           t2 == tcons(r2, tcons(r3, tempty))
6           ------------------------------------ execution-2
7           exists result. rawUnion(t1, t2) == result
8       }
```

Listing 8.9: Example goal from the category *Execution*

To formulate the goal, we use a built-in feature of SPL to introduce four constants r1 through r4 that represent pair-wise distinct rows. SPL provides **local** blocks to limit the scope of such constants, which we employ here for that purpose. We then define an execution goal that introduces two raw tables t1 and t2 and calls rawUnion on them. The name of the goal marks it as an execution goal. Our compiler product line uses this name convention to automatically introduce *ground* requirements for existentially quantified variables like result.

Synthesis The second goal category is dual to the *Execution* category: Here, we explore whether a specifically given result value v is producible via an execution, by asking the ATP to prove that there is a function argument t which produces the result v:

$$\exists\, \mathsf{t}.\ \mathit{ground}(\mathsf{t}) \wedge \mathsf{f}(\mathsf{t}) = \mathsf{v}$$

As before, we are only interested in ground terms t. For our study, we defined 10 synthesis goals that explore different parts of the dynamic and static semantics of SQL. Representatively, we show one goal here that synthesizes a query q and a table store ts such that q is not a value and the reduction of q in ts is stuck:

```
1 goal
2 ------------------------------------ synthesis-4
3 exists ts, q. !isValue(q)
4                    reduce(q, ts) = noQuery
```

Listing 8.10: Example goal from the category *Synthesis*

Again, the name of the goal is used to reveal the goal as a synthesis goal, for which our compiler product line automatically introduces *ground* requirements for existentially quantified variables.

Testing In the third goal category, a user already has an expectation about a concrete input t and output v of a function f and wants to test whether this expectation is met by the specification. This amounts to a quantifier-free proof goal in first-order logic:

$$\mathsf{f}(\mathsf{t}) = \mathsf{v}$$

Here, we rely on the user to make appropriate restrictions about the groundness of t and v (whether the *ground* predicate is required or not may depend on the nature of the particular test). Again, just as for the *Execution* category, our approach allows for testing of specifications that are not directly executable. For our study, we defined 10 test goals that explore different parts of the dynamic and static semantics of SQL. Representatively, we show one goal here that tests whether a selectFromWhere query that selects a column b from a table with columns a and b type-checks with a table with a single column b as expected:

```
1  local {
2    consts a, b : Name
3           ft1, ft2 : FType
4           n : Name
5
6          goal
7          TT == ttcons(a, ft1, ttcons(b, ft2, ttempty))
8          TTC == bindContext(n, TT, emptyContext)
9          sel == some(acons(b, aempty))
10         TT2 == ttcons(b, ft2, ttempty)
11         ---------------------------------------- test-7
12         TTC |- selectFromWhere(sel, n, ptrue) : TT2
13       }
```

Listing 8.11: Example goal from the category *Test*

Verification In the fourth goal category, we consider showing that some property universally holds for a language specification:

$$\forall \mathsf{t}.\ \mathsf{P}(\mathsf{t})$$

We formulated 10 verification goals to ensure properties of the dynamic and static semantics of SQL. Naturally, since we only use first-order logic ATPs, we cannot directly prove arbitrary properties, especially if they require higher-order reasoning, i.e. induction or the application of auxiliary lemmas. One can work around this restriction by explicitly passing axioms which encode necessary lemmas, such as induction hypotheses [Gre+15]. For example, we can prove the inductive step of a theorem stating that intersection preserves typing:

```
1  local {
2          consts RT : RawTable
3
4          axiom
5          rt1 == RT
6          welltypedRawtable(tt, rt1)
7          welltypedRawtable(tt, rt2)
8          rawIntersection(rt1, rt2) == rt3
9          ---------------------------- proof-10-IH
10         welltypedRawtable(tt, rt3)
11
12
13         goal
14         rt1 == tcons(r, RT)
15         welltypedRawtable(tt, rt1)
16         welltypedRawtable(tt, rt2)
17         rawIntersection(rt1, rt2) == rt3
18         ---------------------------- proof-10
19         welltypedRawtable(tt, rt3)
20       }
```

Listing 8.12: Example goal from the category *Verification*

We introduce constant RT as induction variable and provide an induction hypothesis stating that the theorem holds for rt1 == RT. From this, we aim to show that the theorem also holds when adding another row rt1 == tcons(r, RT). The proof of this goal can be derived by a first-order theorem prover, since the necessary induction hypothesis is given as an axiom and hence, no higher-order reasoning is required.

All goals in the *Verification* category are simple goals whose proof does not require the automatic application of induction schemes.

Counterexample In the fifth and final goal category, we aim at finding a counterexample t for a property P as an explanation why the property does not hold:

$$\exists \mathsf{t}.\ \mathit{ground}(\mathsf{t}) \wedge \neg \mathsf{P}(\mathsf{t})$$

Like above, we require that the counterexample t is a ground term and use the name of the goal to automatically introduce *ground* requirements for existentially quantified variables. We defined 10 counterexample goals that disprove statements about the dynamic and static semantics of SQL. For example, we can show that table difference on well-typed tables is not commutative:

```
1 goal
2 ———————————————————————————————— counterexample−6
3 exists rt1, rt2, tt.
4   welltypedRawtable(tt, rt1)
5   welltypedRawtable(tt, rt2)
6   rawDifference(rt1, rt2) != rawDifference(rt2, rt1)
```

Listing 8.13: Example goal from the category *Counterexample*

8.2.2. Automated Theorem Provers

For the purpose of this study, we focus on investigating the performance of automated first-order theorem provers that use saturation-based methods or variants of the sequent calculus to solve problems in first-order logic with equality. We deliberately excluded SMT solvers and provers that use other formats than the standardized TPTP format [Sut10]. We did this in order to keep the results comparable, since other prover formats such as SMT-LIB [BFT16] differ considerably in what constructs are supported, and hence not all the encoding alternatives we describe in Section 8.1 apply.

We considered various theorem provers which competed in the CASC competitions in 2014 and in 2015[2], which all support the standardized TPTP format [Sut10] for automated theorem provers. Out of these, we identified four provers which were able to solve a larger number of our proof goals for at least some compilation strategies: Vampire version 3.0 and Vampire version 4.0 [KV13], eprover [Sch13], and princess CASC version [Rüm08b].

[2] `http://www.cs.miami.edu/~tptp/CASC/24/` and `http://www.cs.miami.edu/~tptp/CASC/25/`

8.2.3. Experimental Setup

We apply the 36 compilation strategies from Section 8.1 to the proof goals from Section 8.2.1 (50 proof goals for the SQL case study, and 50 proof goals for the QL case study). We run all of these input problems on the four theorem provers we selected for our study, which yields a total of 13200 prover calls (and 1200 unsupported calls to eprover when using typed logic).

We run our complete study with a prover timeout of 120 seconds, calling Vampire in CASC mode and eprover in auto mode. We chose this particular timeout after initially trying out several different timeouts, since it yielded the best overall success rates on our example problems for all the four provers we used. A lower timeout was particularly disadvantageous for princess, while a higher timeout did not yield substantially better results for any of the provers. We executed all prover calls on the Lichtenberg High Performance Computer at TU Darmstadt[3]. We used the cluster nodes with Intel Xeon E5-2680 v3 2.5GHz processors, strictly allocating 4 cores and 2GB RAM per core to each prover process.

As a measure of prover performance, we use the success rate of the prover on the given category of proof goals for the timeout of 120 seconds. The success rate for a given goal category indicates how many of the goals in the category the prover could prove within the given timeout. We deliberately excluded both the time to find a proof and the compile time as a measure for prover performance: We observed that the compilation strategies which yield lower execution times for successful proofs are not necessarily the same strategies that also yield high success rates. For the purposes of this study, we decided to focus on investigating how the choice of the compilation strategy affects the overall success rates of the provers.

Note that in the conference version of this paper [Gre+16], we used a different setup for our experiments[4]. We changed the setup to study how changing the available resources affects the results of our experiment. We observed that changing the setup indeed considerably influences the overall success rate of the provers. However, interestingly, we were able to observe the same overall tendencies that we report in Sections 8.3 and 8.4 in both setups, which shows that our main results are reproducible.

8.3. Results of Empirical Study

In this section, we answer the research questions from Section 8.2 with the data from our experiments. We address each research question individually, visualizing the distribution of success rates for different compilation strategies with boxplot diagrams. In the diagrams, we show the results for the SQL and QL problems separately whenever we observe interesting differences between the two case studies. Otherwise, we merge the results of both case studies together in one diagram.

[3] `http://www.hhlr.tu-darmstadt.de/hhlr/index.en.jsp`

[4] Intel Xeon E5-4650 (Sandy Bridge) 2.7GHz processors, allocating 64 cores to each group of calls to one prover (i.e. so that about 64 prover calls in parallel were processed), 2GB RAM per core

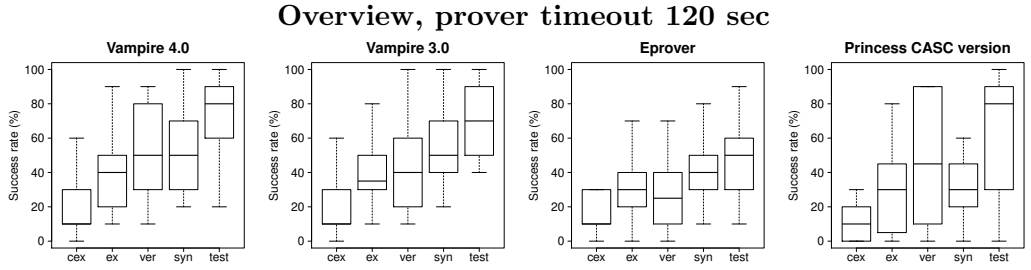


Figure 8.1.: **SQL + QL:** Prover success rates greatly vary with compilation strategy (RQ1).

8.3.1. General Effect on Prover Performance

In RQ1, we ask whether different but equivalent compilation strategies affect prover performance. We evaluate the general effect of different compilation strategies on prover performance by comparing the distribution of success rates for our 36 compilation strategies, separately considering every prover and every goal category. Figure 8.1 visualizes the distribution of success rates for all 36 compilation strategies for the four provers we used, including both the SQL and the QL problems. Each individual boxplot is based on 36 success rates per language specification, one for each compilation strategy we consider - except for the boxplot for eprover, which is based on 24 success rates per language specification, since eprover does not support typed first-order logic as input. We observe that the difference between the smallest and the largest success rate is quite large in every goal category and for every prover, with success rates sometimes even ranging between 0 percent and 100 percent (e.g. Princess, *Test* category).

We conclude that prover performance depends dramatically on the compilation strategy, regardless of the prover chosen and regardless of the goal category used. This observation confirms that it is worthwhile to study the effects of different compilation strategies on prover performance more closely.

8.3.2. Effect of Sort Encoding Strategy

In RQ2, we ask how the strategy for encoding syntactic sorts influences prover performance. We compare the success rates of the three different alternatives for sort encoding against each other across all goal categories: Figure 8.2 visualizes, for each prover, the success rates of our three alternatives for sort encoding. In each sub-figure, we contrast the results for SQL (white) against the results for QL (grey). The individual boxplots are based on 60 success rates. For eprover, we have no data for typed logic (see above). We observe that for both the SQL and the QL problems, the success rates for all strategies that use type guards are significantly lower than the success rates for the other two type encoding strategies, regardless of

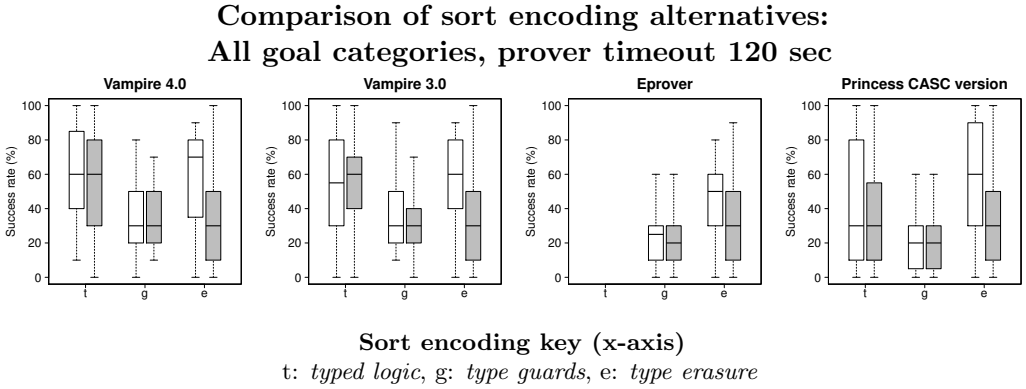


Figure 8.2.: **SQL (white) + QL (grey):** Using type guards for sort encoding significantly lowers prover performance (RQ2).

the prover that was used. When comparing the strategies with typed logic and with type erasure against each other, we observe differences between the SQL and QL problems: For the SQL problems, there is no clear evidence from the data whether typed logic or type erasure offers an advantage. However, for the QL problems and focusing on the two Vampire versions we used, typed logic yields significantly higher success rates than type erasure. We observe the same tendencies if we look at the individual results for each goal category.

We conclude that one should avoid using type guards. A possible explanation for this is that type guards cause an immense blow-up of the formulas. Furthermore, typed logic (if available) seems to be a reasonable choice over type erasure, since it has the potential to improve the overall success rate further at least in some cases. This observation confirms results from similar studies considering sort encodings, such as work by Blanchette et al. [Bla+13b]. We discuss some of this work in Chapter 9.

8.3.3. Effect of Variable Encoding Strategy

In RQ3, we ask how the strategy for encoding variables influences prover performance. We compare the success rates of different alternatives for variable encoding against each other for all categories: Figure 8.3 visualizes, for each prover, the distribution of success rates for each of our four variable encoding alternatives for both of our example specifications together. For Vampire and princess, each boxplot is based on 45 success rates per example specification, for eprover, on 30. We observe that variable inlining and unchanged variable encoding yield higher success rates more frequently than the two naming strategies, although the difference is not significant. Comparing variable inlining and unchanged variable encoding against each other, we observe a slight, but not significant, advantage of inlining for all provers. We observe similar tendencies if we look at the individual results for each goal category.

From the observed tendencies, we conclude that naming strategies should be avoided. Even though the tendencies are not significant, we would recommend

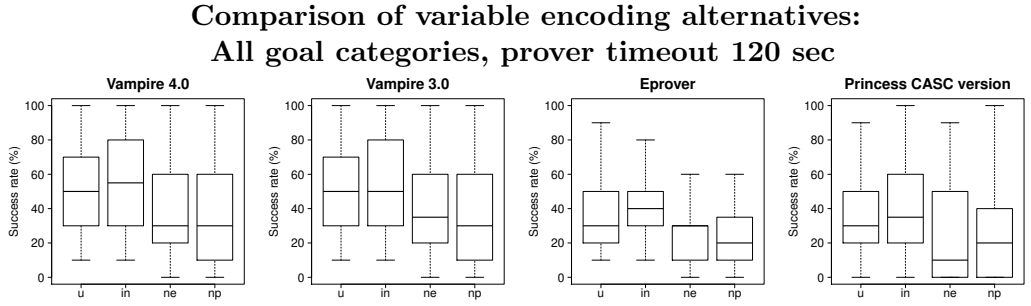


Figure 8.3.: **SQL + QL:** Variable inlining slightly improves prover performance (RQ3).

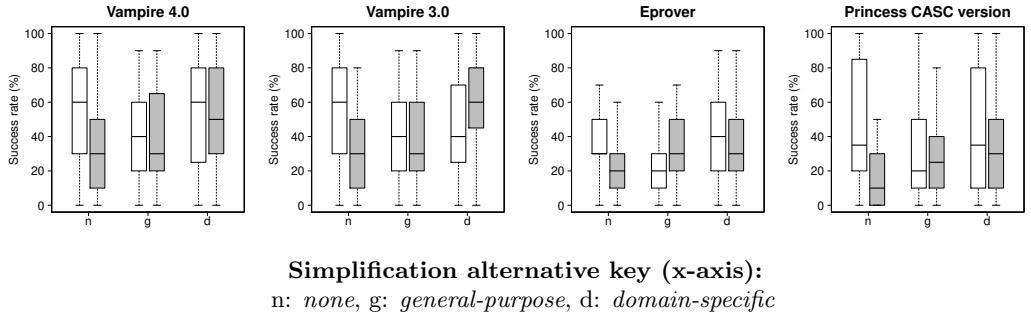


Figure 8.4.: **SQL (white):** Simplification strategies do not significantly influence prover performance; **QL (grey):** Domain-specific simplification yield the highest success rates (but not significantly) (RQ4).

variable inlining as default strategy, since our graphs show that inlining yields success rates that are at least as high as unchanged variable encoding, and occasionally higher.

8.3.4. Effect of Simplification Strategy

In RQ4, we ask how simplifications influence prover performance. We compare the success rates of different alternatives for simplification against each other for all goal categories: Figure 8.4 shows the distribution of success rates for each of our three simplification alternatives. For each alternative, we show the results for SQL and QL in separate boxplots. For Vampire and princess, each boxplot is based on 60 success rates, for eprover, on 40. For both specifications, there is no significant difference between the three simplification alternatives for all provers. One may observe the following non-significant tendencies: For the SQL problems, general-purpose simplification yields the lowest success rates overall. For the QL

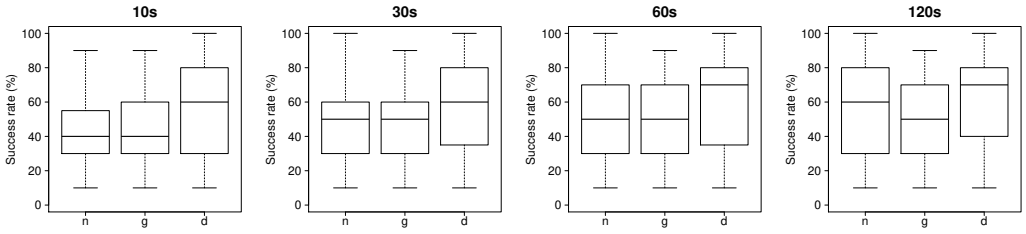


Figure 8.5.: **SQL + QL:** Domain-specific simplifications are particularly advantageous with short prover timeouts (RQ5).

problems, domain-specific simplification yields the highest success rates. We observe similar tendencies if we look at the individual results for each goal category.

We conclude that for our reference timeout of 120 seconds, the simplification strategy does not make a difference.

8.3.5. Effect of Domain-specific Simplification

Our results for RQ4 suggest that, at least for a prover timeout of 120 seconds, domain-specific simplification does not make a clear difference. Therefore, in RQ5, we ask when domain-specific simplification has an influence on prover performance. We experimented with different setups in order to find situations in which domain-specific simplification clearly improves the overall success rate for both case studies. We discovered one such situation, visualized in Figure 8.5: We focus on combinations of simplification strategies with strategies that we already identified as advantageous above. Additionally, we compare the results for different prover timeouts to each other. The figure depicts success rates for the different simplification strategies for all provers together except princess (which yields low success rates for lower timeouts in our problems). Every boxplot is based on 50 success rates per example specification.

We observe that especially for lower prover timeouts, domain-specific simplifications indeed significantly increase prover performance compared to the other two simplification strategies, notably for a timeout of only 10 seconds. However, as the timeout increases, the advantage of domain-specific simplication shrinks. We observe the same tendency in both example specifications separately. We conclude that domain-specific simplification increases prover performance for shorter prover timeouts when combined with other advantageous encoding strategies.

8.3.6. Best Overall Compilation Strategies

In RQ6, we ask whether there is a compilation strategy that performs best for all goal categories, or if not, what the best compilation strategy for each goal category

is. We compare the success rates obtained for each individual compilation strategy across all goal categories and all provers we used: Figure 8.6 depicts two boxplot diagrams (one for the SQL and one for the QL problems) with one boxplot for each of the 36 compilation strategies we investigated. The individual boxplots are either based on 20 success rates (strategies with untyped logic) or on 15 success rates (strategies with typed logic, which is not supported by eprover).

On first sight, the results for the SQL and the QL problems differ from each other. We first look at the results for each set of problems separately.

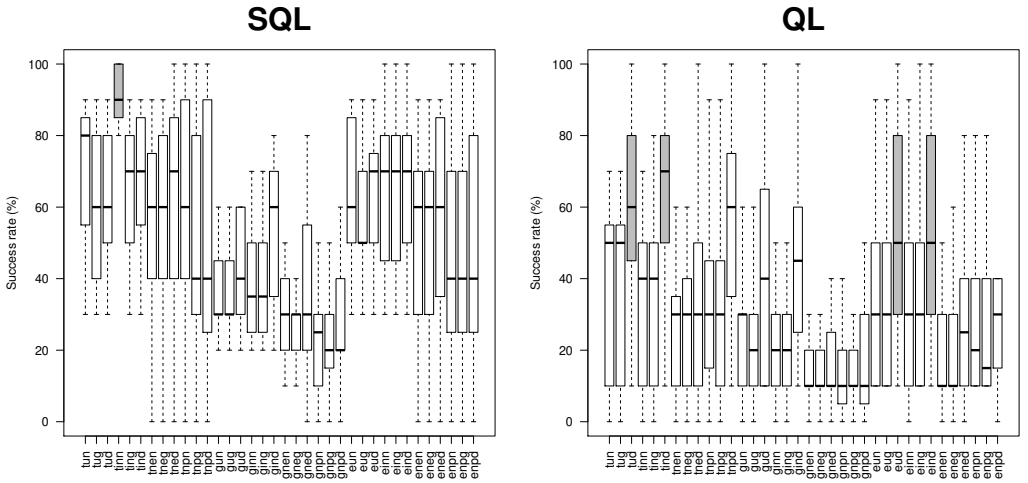
Best overall strategies for SQL We observe that for the SQL problems, the compilation strategy that uses typed logic to encode sorts, inlines variable names, and does not apply any simplification ("tinn", in grey in the SQL graph in Figure 8.6) significantly outperforms all other strategies. Studying our data in more detail (see additional data on artifact page, which is linked below), we observe that this result is mainly due to the two Vampire versions, which both yield very high success rates for "tinn" in all categories for a prover timeout of 120 seconds. Among the strategies that do not use typed logic, there is no clear candidate for which strategy performs best. Eprover, which does not support typed logic, yields the highest success rates with strategy "eud" and also with "eind" (in particular, in category *Testing*).

Looking at the results of individual goal categories and/or lower prover timeouts, we observe that, in particular for lower timeouts, strategies with domain-specific simplification, such as "eind", "eud", or also "tind" are at least as good or even slightly better than "tinn".

Best overall strategies for QL In the QL graph in Figure 8.6, we first observe that there is no single best compilation strategy for the QL problems, but rather four best strategies: "tud", "tind", "eud", and "eind" (marked in grey). These best strategies yield significantly higher success rates than the majority of all other strategies (with the exception of the strategies "tnpd", "gud", and "gind"). Comparing the distribution of success rates between the four best strategies, there is little difference - one could say that there is a slight tendency that the two strategies with typed logic yield higher success rates (but not significantly higher).

Second, the strategy "tinn", which was the best strategy for the SQL problems, yields comparatively low success rates for the QL problems. In general, strategies that use domain-specific simplification seem to have worked better in the QL problems: Here, almost all strategies that use domain-specific simplification yield higher success rates than the strategies without domain-specific simplification. The success rates are particularly high in combination with type erasure/typed logic and inlining/unchanged variable encoding.

Looking at the results for the individual provers in the QL problems, we observe that for eprover, the strategies "eud" and "eind" yield the highest success rates, while for princess, "tud" and "tind" yield the highest rates. In the two Vampire versions, there was little difference between the performance of these two pairs of strategies. Looking at individual goal categories, we observe that in the category *Counterexample*, strategy "tnpd" yields the highest success rates - but only for a



Key for abbreviations

strategy	:= *sort variable simpl.*
sort	:= **t** ‖ **g** ‖ **e**
	t: typed logic, **g**: type guards, **e**: type erasure
variable	:= **u** ‖ **in** ‖ **ne** ‖ **np**
	u: unchanged, **in**: inlining, **ne**: naming everything,
	np: naming of function parameters and results
simpl.	:= **n** ‖ **g** ‖ **d**
	n: none, **g**: general-purpose, **d**: domain-specific

Figure 8.6.: **SQL**: Prover success rates are best for typed logic (if available) and inlining; **QL**: Prover success rates are best for strategies with domain-specific simplification, and no naming strategies or type guards (RQ6).

timeout of 120 seconds. With lower prover timeouts, "eind", "eud", "tind", and "tud" yield higher success rates in category *Counterexample*. In categories *Testing* and *Verification*, "eind" performs best.

Considering our two case studies together, we conclude that there is no single best compilation strategy for all goals. This conclusion differs from the conclusion in our conference paper [Gre+16], where we based our conclusions solely on the observations for the SQL problems. However, also when taking the results for the QL problems into account, the most advantageous strategies remain the ones that combine our findings for the previous RQs: strategies with either type erasure or typed logic, and either variable inlining or unchanged variable encoding, plus, in some cases, domain-specific simplification.

8.3.7. Discussion

Firstly, our results empirically confirm that different compilation strategies can have a huge effect on prover performance - even if the strategies only produce subtle differences in the encoded problems, and even if the strategies apply optimizations which overlap with what ATPs may do internally. This first result is very likely to hold beyond our two case studies and our exploration proof goals, due to the heuristic nature of ATPs.

Secondly, our results show that there is no single best strategy for all input problems: Rather, one should try different compilation strategies with a new problem to identify which one works best.

Despite this general observation, we observed the same general tendencies for advantageous compilation strategies in both of our example specifications: typed logic or type erasure in combination with either variable inlining or unchanged variable encoding. In certain cases (e.g. for shorter prover timeouts), domain-specific simplification also helps to increase the success rate. These tendencies mostly correspond to the observations in our earlier conference paper [Gre+16]. Hence, we were able to show that our overall results carry over to a different language specification as well as to a different hardware setup. This and the differences between the two example specifications increases our confidence that the tendencies we report will also carry over to other language specifications with different exploration goals, and to other hardware setups.

Based on our current observations, we make the following recommendations for new exploration goals on other language specifications: When using provers that support typed logic, try combining typed logic, variable inlining, and domain-specific simplification first. With provers that do not support typed logic, combine type erasure, variable inlining, and domain-specific simplification. Experiment with slight variations of the compilation strategy, such as omitting simplification.

The complete data from our study is available at `http://www.st.informatik.tu-darmstadt.de/artifacts/comp-fol-study-journal/`: all compiled input problems, the complete logs of all provers on the problems, result summaries, and additional graphs compiled from our raw data that we did not show here.

8.4. Domain-Specific Axiom Selection

So far, our goal was to compare compilation strategies from SPL to first-order logic. To focus on effects of the compilation strategies, we included exactly the same axioms in each goal category (except if a goal required an additional axiom such as an induction hypothesis), namely *all* the axioms that we generate from a language specification[5].

However, not applying any axiom selection can of course yield unnecessarily large input problems, since the problems may include potentially irrelevant axioms. These irrelevant axioms may influence the internal heuristics and search algorithms of a prover, and hence the overall success rates. We extended our previous experiments from Section 8.2 with chosen domain-specific strategies for axiom selection, in which we exploit the fact that we know which classes of axioms our problems contain. The additional research question we study in this section is

RQ-Ax How does domain-specific axiom selection influence the success rates?

8.4.1. Selection Strategies

We implemented two independent domain-specific strategies for axiom selection. Based on these two selection strategies, we extended our compiler product line (see Section 8.1.4) with four different variants for domain-specific axiom selection:

1. *Select all* This strategy uses all generated axioms, like previously. We included this variant for comparison purposes.
2. *No inversion axioms* We omit the automatic generation of inversion axioms for total functions (see Section 5.4.2). This omission reduces the size of all input problems significantly, since the inversion axioms we generate typically produce large disjunctions (the larger the original function, the larger the inversion axiom). While inversion axioms are needed for some proof problems on language specifications (e.g. when trying to prove steps from proofs of type soundness using automated theorem provers, which we explored in a previous paper [Gre+15]), we observed that the inversion axioms are not necessary for proving most of the exploration goals that we study in the present article.
3. *Select reachable* We conservatively determine which axioms from the original axiom set are *reachable* from a goal, and discard all other axioms. We say that an axiom ax is *directly reachable* from the proof goal g or from another axiom ax' if
   a) the axiom ax is part of the definition of a datatype used in g or in ax' or if
   b) the axiom ax is part of the definition of a function used in g or in ax'.

   For example, consider a goal g that uses the total function f. Let us assume that the function equations of the definition f were compiled into three axioms and one inversion axiom (following the scheme from Section 5.4.2). Then all three

[5] For categories that did not include proof goals on the type system (e.g. *Execution*, since typing judgments are not "executed", but only check expressions against certain types), we left out the axioms generated from the type system specification.

axioms for the function equations and the inversion axiom are *directly reachable* from g. Moreover, let us also assume that g uses a constructor C of a datatype T at some point. Datatype T consists of two constructors. Compilation of the datatype T to first-order logic with the scheme from Section 5.4.1 then yields two injectivity axioms (one per constructor), one axiom stating the difference of the two constructors, and a domain axiom. All of these axioms would as well be *directly reachable* from g.
We determine the set of *reachable* axioms by iterating the notion of *directly reachable* starting from the proof goal and continuing over every new axiom which we include until we reach a fixed point.
Note that from our compilation scheme, we know exactly which axioms define a datatype or a function, hence we can directly select the correct axioms, exploiting our domain knowledge of the structure of each problem. We do not select among the axioms that define a datatype or function, since this might introduce unsoundness into the problem description. Additionally, we also conservatively include any user-defined axioms into the input problems, assuming that these axioms always contain relevant information. In our two case studies, user-defined axioms are, for example, the typing rules and induction hypotheses.

4. *No inversion axioms + Select reachable* This strategy combines the two previous strategies: We first omit inversion axioms and then use the remaining axiom set as a basis for determining the axioms that are *reachable* from a proof goal. Hence, this strategy omits the largest number of axioms compared to the previous three strategies.

We discuss related approaches for axiom selection, such as SInE [HV11], in Subsection 8.4.3 and in Chapter 9.

The number of axioms ruled out in our example problems by the selection strategies from above greatly differs for each different goal and each different strategy. The number of axioms ruled out by selection strategy "no inversion axioms" alone is more or less constant[6] and only differs for the two different example specifications: The strategy ruled out about 14 percent of the axioms from the QL problems, and about 17 percent of the axioms from the SQL problems. Among the axioms ruled out are also the largest axioms in the axiom set. The number of axioms ruled out by selection strategy "select reachable" greatly differs for every goal. Since this strategy operates conservatively, it selects large parts of the overall axiom set, but never the entire set for the goals that we considered. Selection strategy "no inversion axioms + select reachable" combines the two previous strategies and hence rules out the addition of axioms ruled out by each individual selection strategy.

8.4.2. Comparison results

We compare the distribution of success rates for each of the four strategies for axiom selection from the previous section. Figure 8.7 depicts the distribution of

[6] Small differences between different goals occur since some goals require additional axioms for induction hypotheses or ground predicates that are not included in all problems.

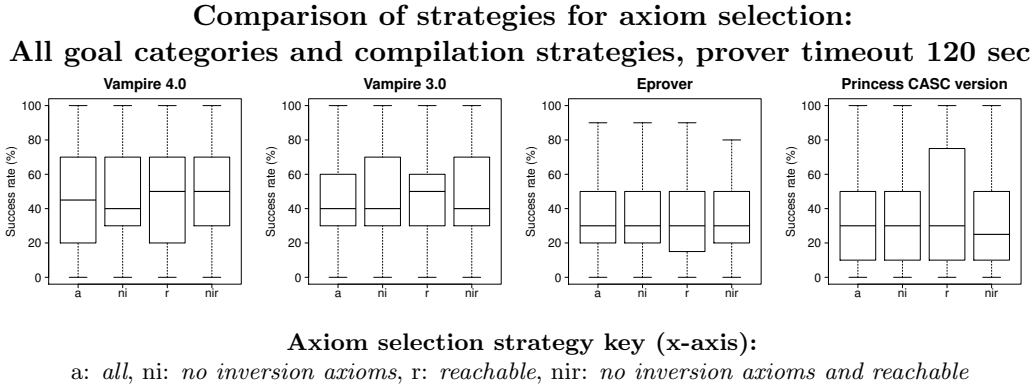


Figure 8.7.: Strategies for axiom selection make almost no difference (RQ-Ax).

success rates for the four provers we used, across all compilation strategies, all goal categories, and for both the SQL and QL problems from our previous study (see Section 8.2). Every boxplot is based on 360 success rates (eprover: 240).

We observe that there are only insignificant differences between the four strategies for axiom selection. This observation neither changes for lower prover timeouts, nor when focusing on comparing only success rates for the compilation strategies that we identified as advantageous in Section 8.3, nor for individual goal categories. Considering just the SQL problems, the *select all* strategy yields slightly higher success rates for a timeout of 120 seconds. For prover timeouts lower than 60 seconds, strategies *No inversion axioms* and *No inversion axioms + Select reachable* sometimes have a slight advantage. For the QL problems, *No inversion axioms* and *Select reachable* often yield higher success rates. However, all of these differences are not significant.

The graphs which show the distribution of success rates for individual combinations of specific compilation strategies and strategies for axiom selection look very much like the graphs from Figure 8.6 for RQ6, roughly replicating each individual boxplot four times. Hence, our strategies for axiom selection did not influence the general individual tendencies of the compilation strategies.

We conclude that the domain-specific strategies for axiom selection we considered have no influence on the success rates, at least not for specifications as large as our specifications of SQL and QL (SQL: ca. 280 axioms, QL: ca. 360 axioms).

The complete results for the comparison of domain-specific strategies for axiom selection, including additional graphs, are also available on `http://www.st.informatik.tu-darmstadt.de/artifacts/comp-fol-study-journal/`.

8.4.3. Discussion

In our extended study, we demonstrated empirically that applying domain-specific selection strategies hardly influences the overall prover success rates.

One possible explanation for this result is that automated theorem provers might themselves be good enough at figuring out which individual axioms from our input

problems are relevant for solving the problem: Many modern automated theorem provers implement themselves general-purpose heuristics for selecting relevant axioms beforehand. For example, some of the strategies used by the Vampire CASC mode use a system called SInE [HV11] to apply axiom selection. SInE employs a *symbol-based* selection scheme which iteratively selects axioms that share functions or constants with the previously selected axioms, starting from the proof goal. A naive implementation of SInE selection would select about the same axioms as our *Select reachable* strategy (possibly discarding some of the user-defined axioms that we never discard in some problems in addition). However, in practice, the SInE implementation used in Vampire contains various general-purpose heuristics which may make this selection more or less generous than our *Select reachable* strategy. Our observation from Section 8.4.2 seems to indicate that these general selection heuristics implemented within the provers are always at least as good as our domain-specific selection strategies.

A second, additional explanation for our result could be that our case studies are not large enough for our selection strategies to make a difference for the provers. That is, both before and after axiom selection, the overall size of our proof problems is small enough such that even a larger number of irrelevant axioms would not affect the provers' success rate. Since provers like Vampire and Eprover are able to solve problems with thousands of irrelevant axioms (e.g. in the LTB (large theory batch problems) division of the CASC competition for automated theorem provers), this explanation seems very likely.

Summarizing our results from this and the previous section, we conclude that the question of how to compile a given problem to first-order logic is far more relevant to prover success than the question of how to optimize axiom selection. Hence, adding for example specialized axioms that may only help the provers in a few specific cases is much less problematic than choosing a not so advantageous compilation strategy.

Limitations In our study, we deliberately focused on small, special-purpose DSLs, such as the ones found in industry, in order to first understand the effects of compilation strategies on small languages. Since our two example specifications (SQL and QL) are quite different languages, it is likely that our results carry over to other languages of similar size and complexity. As for larger and more complex languages (e.g. general purpose languages such as Featherweight Java [IPW01b]), it is not clear whether our results regarding compilation strategies and domain-specific axiom selection strategies carry over: Such language specifications would both have more complex axioms and also larger axiom sets, which would probably trigger different internal heuristics within the ATPs.

8.5. Summary

We presented an empirical case study for comparing different encoding strategies for TPTP with regard to the success rate of provers. We discovered which combinations of encoding strategies yield slightly better success rates over others. Additionally,

we conducted an orthogonal study on axiom selection strategies on the user side. We learned that such selection strategies do not influence the success rates of provers.

All of the results presented within this chapter were used when developing our automated proof strategies for type soundness proofs of DSLs and in particular, for obtaining the verification results within our case studies we presented in Chapter 7. The results of our empirical study may serve as guidance to domain experts who want to automate other verification domains using VeriTaS and need to develop their own encoding strategies.

Chapter

9

Related Work

We compare the different contributions of this thesis against existing relevant work, focusing on three different overall topics: Firstly, we discuss other work regarding the mechanized verification of *type soundness* in general, including of course mechanized proofs, but also any other mechanical approaches to the problem, such as for example model checking techniques (Section 9.1). Secondly, we discuss existing verification systems and verification infrastructures and compare them to our VeriTaS infrastructure that we presented in this thesis (Section 9.2). Finally, we discuss different works in the area of developing and comparing different encoding strategies of proof problems for automated theorem provers (Section 9.3).

9.1. Mechanized verification of type soundness

In this thesis, we focus on how to automatically obtain mechanized *proofs* of type soundness. *Proving* type soundness logically and in a mechanized way is a common and dominant technique in the area of mechanized verification in research. Therefore, we first discuss existing approaches to obtaining mechanized proofs of type soundness for different languages (Subsection 9.1.1), focusing on the degree of automation the different approaches achieve.

However, one may also interpret term “verification” in a broader sense, not necessarily meaning full proofs or maybe not even proofs at all. There are approaches in the literature that enable the lightweight mechanization and exploration of type system specifications, allowing for discovering soundness problems via testing. We discuss these approaches in Subsection 9.1.2. Finally, there are also completely different approaches for the mechanized verification of type soundness, ranging from model checking approaches to type systems that are sound by construction. We give a brief overview of these approaches in Subsection 9.1.3.

9.1.1. Mechanized proofs

In Section 1.1, we gave an overview of a number of mechanized type soundness proofs from research, notably on the work achieved in the context of the PoplMark challenge. In that section, we focused on the overall size of such mechanizations and on the skill a developer needs to create a mechanized type soundness proof with different existing tools. We now take up this discussion again, focusing on the degree of automation of various mechanizations and comparing them to our work.

Firstly, neither the larger mechanized type soundness proofs we mentioned in Section 1.1 ([FV06; KN05; LCH06]), nor the existing solutions to the PoplMark challenge [Ler07; Ber12; Vou12; CS12] achieved full automation - this also not having been the goal of these works, notably in the context of PoplMark. To date, no one has presented an automated solution to the PoplMark challenge. In Subsection 3.1.1, we discussed one of the main reasons that complicates the mechanization of type soundness proofs: the "name-binding problem".

The degree of automation within the existing solutions to the PoplMark challenge using interactive theorem provers such as Isabelle [Ber07] and Coq [Ler07; Vou12] is rather low: Most proof steps need to be spelt out explicitly by the developer. In Isabelle, there is good support for automating the proofs of low-level steps, via Sledgehammer [BP16], which we also demonstrated in Section 3.2. However, developers definitely have to spell out all of the necessary high-level steps, which requires a certain expertise in using Isabelle. In particular, dealing with the "name-binding problem" requires a lot of expertise, as we argued in Subsection 3.1.1. But also when focusing on languages without first-class binders, as we do in this thesis, Isabelle proofs require the manual specification of lots of steps, as we demonstrated in Section 3.2

Further automation for generating type soundness proofs in interactive theorem provers could theoretically be achieved by developing automated tactics using the tactic languages provided by Isabelle [MMW16] and Coq [Del00; Zil+13], as we discussed in Subsection 3.4.3. However, we know of no previous work which used one of these tactic languages to achieve a higher degree of automation for type soundness proofs. In any case, such tactics would have to be developed by system experts, who might not necessarily have inside domain knowledge about type soundness proofs.

In contrast to and inspired by the work we just mentioned, we propose in this thesis to focus any automation efforts for type soundness proofs on DSLs without constructs for abstraction and name-binding (so that substitution is required - see Subsection 3.1.2). Furthermore, the VeriTaS verification infrastructure we propose is deliberately designed so that the entry point for domain experts for developing automated proof strategies is low: VeriTaS is designed as an extensible Scala library and provides numerous reusable components for proof automation (input formats, basic tactics, some higher level strategies, connections to existing automated theorem provers).

As for the degree of automation achieved for type soundness proofs in comparison to the respective proofs in Isabelle/HOL (in which we conducted a complete example soundness proof for comparison in Section 3.2) and Coq (in which we did not

explicitly conduct an example proof in this thesis), our automated proof strategies for this verification domain are able to generate very large parts of type soundness proofs automatically: Only a small number of user annotations and auxiliary lemmas is needed as input for the generation of the proof graphs, and a handful of low-level problems for manual inspection remain. Overall, generating type soundness proofs using VeriTaS requires less special skills in using a verification system as using an interactive theorem prover: A basic understanding of type soundness proofs as well as Scala knowledge is sufficient.

The probably highest potential of full automation among the set of solutions submitted to the PoplMark challenge is the Twelf approach [HL07]. As previously mentioned, Twelf is a special-purpose theorem prover for properties of logics and programming languages based on the logical framework (LF). In particular, Twelf provides an elegant approach to the name-binding problem (HOAS). As for proof automation, Twelf provides an interactive proof mode as well as support for automated inductive theorem proving [SP98], using a meta-logical framework. This support does not only target type soundness proofs, but proofs of properties of a certain format in general (in the form of $\forall ... \exists ....$).

Schürmann [SP98] demonstrates the automation approach of Twelf at the example of type preservation for ML. The degree of automation achieved by this support is in principle very close to the degree of automation achieved by our automated domain-specific proof strategies for type soundness proofs within VeriTaS: Developers only have to specify auxiliary lemmas. But in addition, any points within the proofs where auxiliary lemmas have to be used also need to be given by a developer, whereas our automated proof strategies in VeriTaS generate lemma application steps for type soundness proofs.

The most crucial difference between our approach and Twelf is that in Twelf, encoding a type system specification and a corresponding soundness proof requires thorough knowledge of logical frameworks, as we also discussed already in Section 1.1. In this thesis, we target domain experts (see Subsection 1.3.2), who do not necessarily have this rather special knowledge. To avoid the logical frameworks notation, the LF-based tool SaSyLF [ASS08] allows for specifying language syntax, semantics and type systems in Twelf by using paper-like notation. However, our inspection revealed that SaSyLF targets an educational context for teaching students type theory, and is not suitable for the development of DSLs. Finally, while Twelf shines when encoding construct for abstraction (due to the HOAS approach), it is unclear how to encode the often rather concrete and low-level language constructs of DSLs such as SQL within Twelf.

We now consider a small selection of mechanized type soundness proofs that were not developed in the context of the PoplMark challenge.

Syme and Gordon present a semi-automated technique for type soundness proofs of virtual machines that do not involve inductive reasoning [SG02] (e.g. of the Spark bytecode language). Their approach requires that a user indicates relevant reduction rules to control for example the unwinding of recursive definitions in the proof. This *guided reduction* serves as input to a decision procedure. Concretely, users need to specify a set of problematic predicates or functions from a specification, together

with how and where to apply the fundamental rules associated with these predicates. This is in principle similar to our specification of user annotations for functions and associated lemmas. However, our approach also targets type soundness proofs that involve inductive reasoning, which is important for being able to address a number of interesting DSLs.

In this thesis, we also conducted a type soundness proof of a DSL within Dafny [Lei10] (see Section 3.3). Even though Dafny is originally meant as a programming language with verification support to aid the verification of imperative programs, developing these proofs worked well and we achieved a high degree of automation: For our example proof, we only had to specify the auxiliary properties needed as well as manually designate the points within the overall proof where such properties needed to be used. Except for that latter point, the degree of automation is comparable to the one we achieve with our automated proof strategies.

However, in contrast to Dafny, our strategies also generate lemma application steps. Also, an important difference between our approach and Dafny is that our proof strategies generate a human-readable proof that users may fully inspect to convince themselves of the overall correctness of the proof if they do not trust in the tools used. In Dafny, the automatically derived parts of a proof remain hidden to users.

As for Dafny's general suitability for the automation of other verification domains, the same arguments as previously apply (see also Subsection 3.4.3): Such automation would require insider knowledge about Dafny, by using a tactic language such as Tacny [GT16].

Mechanized Type Soundness Proofs of DSLs As for mechanized type soundness proofs of DSLs, which is the primary focus of this thesis, there unfortunately is little related work to be found. There is extensive work in the area of language workbenches for creating DSLs that contains support for specifying and validating type systems of DSLs in general [Erd+13; Erd+15]. One famous example of a language workbench is Xtext [EB10][1]. Bettini and Völter developed a language for specifying type systems for languages developed in Xtext [Bet+12]. Jung et. al. develop an approach for implementing type checkers for their DSLs in a structured way [JSH13]. Another example of language workbenches are MPS [VP12], Spoofax [KV10], and SugarJ [ER13], which also include support for defining and implementing type systems for DSLs. However, to the best of our knowledge, none of these approaches specifically attempts to support the mechanized verification of the type system's soundness. One exception to this is Spoofax, with an ongoing project on supporting verification of a language's properties from within the language workbench [Vis+14]. In principle, one could link our support for automated type soundness proofs of DSLs to one or more existing language workbenches.

As we have seen in Chapter 3, the mechanization of type soundness proofs of the DSLs we consider in this thesis is cumbersome with existing theorem provers, but conceptually rather uninteresting and straightforward. We suppose that this is the

[1] `http://www.eclipse.org/Xtext/index.html`

reason why there is little work to be found on mechanized type soundness proofs for DSLs. This observation confirms again the motivation for the language focus in this thesis: Our work has the potential of raising the number of mechanized type soundness proofs for DSLs.

9.1.2. Lightweight mechanization and exploration

PLT Redex [Kle+12a] provides a lightweight specification and exploration environment for programming languages. Redex can visualize test executions and offers randomized testing support for checking behavioral properties. As such, Redex may also be used to verify type soundness, by exploring specifications of type systems and testing them. Numerous developments in Redex use standard specifications type systems as examples.

In principle, one could use any existing automated testing approaches for the lightweight exploration of type systems specifications and for verifying, to some degree, their soundness. For example, one could specify a type system within Scala and use Scala's property-based testing library ScalaCheck [2]. However, we are not aware of any scientific work regarding how one can, for a certain set of languages, use a testing-bases approach to verify type soundness automatically.

Testing-based approaches are particularly useful during an early stage of the development of a type system, allowing to detect simple typos and other common human mistakes. Since VeriTaS is implemented as a Scala library and since in particular our specification language for type systems ScalaSPL is a subset of Scala, our approach can easily be combined with testing-based approaches such as ScalaCheck to find errors prior or parallel to generating type soundness proofs. However, note that testing can never guarantee the total absence of any errors.

A more specialized approach to the verification of type soundness was developed by Lorenzen and Erdweg [LE13], building on PLT Redex. This approach focuses on automatically verifying the soundness of syntactic language extensions, assuming type soundness of a base language that is extended with syntactic sugar. Concretely, Lorenzen and Erdweg automatically rewrite typing rules for syntactic extensions to a typing derivation that uses only language constructs from the base language. Thereby they can prove type soundness for the extended language assuming the base language is sound. Their approach is well-suited to complement any approach that attempts to generate type soundness proofs, such as ours. However, it can neither be used to prove type soundness for a language from scratch, nor to obtain an actual type soundness proof. In contrast, our automated proof strategies for generating type soundness proofs for DSLs generate full type soundness proofs from scratch.

The meta-theory tool Ott [Sew+10] is a lightweight metalanguage for specifying programming languages. Additionally, it offers consistency checks of specifications and can translate specifications to code for various proofs assistants (among them, Isabelle [NPW02], Coq [Tea19], and Twelf [PS99]). As an input language, Ott uses a syntax for specifications of type systems that is quite close to the syntax used in

[2] `https://www.scalacheck.org`

papers, hence the entry level for domain experts with relatively little knowledge in interactive theorem provers is low. However, the automated translations of Ott to various interactive theorem provers only ever generates proof *stubs* with more or less large gaps. To fill them, a domain expert would again require advanced skills within the chosen interactive theorem prover. In contrast, our automated proof strategies for type soundness proofs in VeriTaS automatically generate sub-problems which may again be displayed in the input format, ScalaSPL. Also, the generated proof graphs may be manually refined *within* VeriTaS. Hence, VeriTaS enables users to avoid the direct interaction with other provers.

9.1.3. Other Approaches to Verification of Type Soundness

Roberson et al. [Rob+08] present an automated approach for verifying the soundness of type systems using a software model checker: They systematically generate every type-correct intermediate program state (pruning the search space efficiently by detecting similarities), let the state take a step, and then attempt to type the result. Thus, they are able to systematically detect soundness bugs in a type system specification automatically. Roberson et al. show that this approach is feasible for checking the type soundness of several example languages.

In principle, this approach is similar to testing-based approaches, but is a little more systematic (since the approach aims at covering the entire search space and uses model checking techniques to do so). However, the main disadvantages when compared to approaches that generate soundness proofs remain: The overall absence of soundness problems cannot be guaranteed, and no human-readable argument that type soundness holds is produced. Again, an approach such as the one by Roberson et al. could very well complement our approach to generating type soundness proofs.

Cimini et al. [CMS16] present an approach for automatically certifying that the specification of a type system is sound by checking it against a meta type system. The meta type system formalizes certain general rules about how a type system has to be constructed in order to satisfy progress and preservation. Cimini et al. formally prove that any specification of type systems that can be successfully checked against the meta type systems is sound. For languages which can successfully be type-checked against the meta type system, the implementation developed by Cimini et al. could generated a proof of type soundness that can be machine-checked by the Abella proof assistant [Gac08].

Theoretically, the approach by Cimini et. al. is very interesting since it has the potential of making the generation of type soundness proofs superfluous altogether. However, it is not straightforward how to use the approach on arbitrary languages, notably on DSLs such as for example the subset of typed SQL that we used in this thesis. The input syntax used by the approach of Cimini et al. resembles the input syntax of Twelf/logical frameworks. This syntax requires a certain familiarity with logical frameworks that domain experts typically do not possess (see discussion above).

9.2. Verification infrastructures and theorem provers

We discuss other existing verification infrastructures and theorem provers that one could in principle use for automating type soundness proofs and compare them to the verification infrastructure we propose (VeriTaS). We focus the discussion on how well other systems meet our requirements for an infrastructure that is well-suited for automating domain-specific verification tasks (see Section 4.1).

Note that most of the systems we discuss in this section are systems that target general-purpose verification and/or general program verification. To the best of our knowledge, our idea of providing a verification infrastructure that specifically targets the automation of different verification domains using domain-specific formats for input specifications as well as for implementing verification strategies is novel. We are not aware of a similar project with this goal. At the end of this section, we discuss verification systems that target specific verification domains other than our target verification domain.

9.2.1. Interactive theorem provers and tactic languages

We first compare our verification infrastructure against existing interactive theorem provers with tactic languages.

In the interactive theorem prover Coq [Tea19], tactics for constructing proofs can either be written in OCaml, in the internal tactic language ltac [Del00], or in the dependently-typed and more recent internal tactic language Mtac [Zil+13] (a monad for typed tactic programming in Coq). In the interactive theorem prover Isabelle [NPW02], tactics for constructing proofs can either be written in Isabelle/ML, or via a recent collection of tools for a "proof method language" called Eisbach [MMW16], which allows for defining proof methods via Isabelle's Isar syntax [Wen02]. For Dafny [Lei10], there is a tactic language called Tacny [GT16].

The tactic languages just mentioned differ in how one can express and combine tactics and in which higher-order syntax constructs one may use for programming a tactic. However, they all have in common that they only allow for inspecting and querying the current goal state within a proof, and then manipulate that state by applying other available tactics to it. In general, tactic languages do not allow for querying the AST of a problem specification in order to for example inspect the different cases of a function definition.

Most importantly, existing tactic languages do not allow for the approximate construction of subgoals or auxiliary lemmas: Any intermediate goal can only ever arise from the successful application of the tactics from before. That means existing tactic languages cannot lay out an approximate *proof structure*, but only provide a plan for the *steps* to be executed to prove a goal.

In the verification infrastructure that we propose in this thesis, we deliberately take a different view on proof automation: We focus on representing a proof structure by explicitly forcing the generation of approximate subgoals and putting them into the center of how we represent a proof structure. The steps that have to be taken to get from one generated subgoal to another link these subgoals. This different

view enables a complete decoupling between the generation of an approximate proof structure and the verification of individual steps within the proof, which we argued for in Section 4.1.

Furthermore, the substantial difference between our verification infrastructure and the interactive theorem provers mentioned is that we explicitly target domain-specific verification, supporting different domain-specific input formats. In constract, existing provers are general-purpose theorem provers, supporting a single, general-purpose input format.

9.2.2. Systems With Automated Provers

There are numerous existing systems that employ tools and techniques from automated theorem proving to support the automated verification of individual proof steps, just like the VeriTaS verification infrastructure that we propose in this thesis.

System discussed previously Isabelle provides Sledgehammer [BP16], which allows for using a number of ATPs and SMT solvers within Isabelle for discharging subgoals. We demonstrated and discussed Isabelle Sledgehammer in Chapter 3. Dafny [Lei10] internally calls the SMT solver Z3 [DB08] via the intermediate verification language Boogie 2 [KR10] for automating a large number of proof steps, as we also demonstrated in Chapter 3. We list both systems here for completeness, but otherwise refer to the previous sections which discuss in detail how VeriTaS is related to Isabelle and Dafny.

Why3 Why3 [Bob+11] is a software verification platform that provides a frontend to a vast number of third-party theorem provers, including SMT solvers, ATPs, as well as interactive theorem provers. In an abstract way, the overall architecture of Why3 is very similar to that of our verification infrastructure VeriTaS and partially served as inspiration for our approach: Why3 is implemented as an OCaml programming library so that Why3 users may easily attach their own projects to Why3 by using the library. For similar reasons, we implemented VeriTaS as a Scala library.

However, one of our main goals was to support domain-specific verification, specifically allowing developers to attach domain-specific input formats. Hence, VeriTaS is generic in an input format, in contrast to Why3, which has one single input language. This goal of supporting domain-specific verification also motivated our language choice for the implementation of VeriTaS, since Scala offers very good support for developing embedded DSLs.

In Why3, users formulate *proof tasks* to verify properties. These proof tasks are then gradually translated into the input format of supported provers via a series of *transformations*. This approach corresponds to the approach we used to compile the VeriTaS input format (SPL) to first-order logic: We defined a series of transformation steps, partially with different encoding strategies of individual features, that we combine to obtain overall encoding strategies.

One crucial difference between our VeriTaS verification infrastructure and Why3 is our concept of proof graphs for composing and visualizing the single proof problems

that we generate as an overall proof to a problem. Why3 has no similar structure.

Leon Leon [Bla+13a] is a verification system for a subset of the Scala programming language. One can compare Leon to Dafny in that it is actually a programming language, but allows for augmenting function definitions and other language constructs with pre- and post-conditions. In the background, Leon uses a number of ATPs and SMT solvers (mostly Z3 [DB08]) for attempting to automatically prove these conditions. Additionally, there is support for translating from Leon to Isabelle [HK16] and for using Isabelle as another verifier in the background. For developing our domain-specific input format for specifications of type systems, we took inspiration from Leon's syntax (especially regarding the usage of `require` and `ensuring` for formulating properties). The subset of Scala that ScalaSPL supports is intentionally smaller than the subset of Scala that Leon supports: We focus on only being able to express specifications of type systems in ScalaSPL. Additionally, ScalaSPL allows for augmenting a specification with proof-relevant, domain-specific user annotations, which Leon does not support.

The main difference between Leon and our VeriTaS verification infrastructure is that VeriTaS targets the automation of domain-specific verification tasks by domain experts, while Leon targets the verification of general-purpose programs.

KeY KeY [Ahr+16] is a system for verifying the functional correctness of (sequential) Java programs. KeY uses the Java Modeling Language for specifying properties of programs. The system internally employs a sequent calculus for Java Dynamic Logic, an adaptation of Dynamic Logic [HTK00]. It is also possible to connect external SMT solvers for closing proof obligations. The sequent calculus used within KeY specifically targets the verification of sequential Java programs, which allows for achieving a high degree of automation: KeY is able to automatically verify functional correctness properties for correctly specified, sequential Java programs. For example, the greatest KeY-based case study so far for verifying the TimSort algorithm [Gou+] for sorting reported that at least 99% of the necessary rule applications could be closed automatically.

KeY was designed for verifying that a particular program satisfies a property for all possible input values. KeY does not target the verification of properties that range over a set of programs, such as, for example, type soundness. Probably, KeY's internal calculi could be adapted for this purpose as well, but this would require expertise regarding the internals of the KeY system as well as engineering effort.

Like our VeriTaS verification infrastructure, KeY emphasizes that users should be able to explore the intermediate proof state. We build a proof graph that contains as nodes intermediate sub-obligations. After triggering verification of the proof steps within a proof graph, we also save the verification result of each step within the proof graph. Users may inspect and modify a proof graph, for example to refine steps with inconclusive verification results. KeY builds a proof tree to enable user inspection and interaction, visualized like a directory structure. Such a proof tree contains one node for each single transformation step of a program that occurs

during verification. Users may inspect how each step in the proof tree modifies the program.

9.2.3. Domain-specific verification systems

Next to the general-purpose verification tools that we discussed above, there is a number of verification systems and platforms that target a specific verification domain. We discuss an example of such a system.

EasyCrypt [Bar+11] is an automated tool for verifying properties of cryptographic systems. It allows for structuring cryptographic proofs using a standard technique from the area: game-playing. This technique organizes cryptographic proofs as sequences of interactions between adversaries and oracle systems. An oracle may for example allow an adversary to retrieve a plaintext message from a cryptographic protocol that we want to prove secure. The "game" then consists in using this oracle for "breaking" a mathematical cryptographic primitive that is known to be hard to break. That is, the technique aims at reducing the problem of breaking a cryptographic protocol to the problem of breaking a cryptographic primitive mathematically known to be "secure". If this reduction succeeds, the cryptographic protocol is said to be as secure as the cryptographic primitive in question.

EasyCrypt uses a domain-specific input format for specifying such "games". For their machine-checked verification, the tool implements a number of complex special-purpose reasoning techniques, e.g. logical relations between the games using probabilistic Relational Hoare Logic (pRHL) and information-theoretic reasoning. Similar to VeriTaS, it translates individual verification conditions to first-order logic and calls external ATPs and SMT solvers to verify them.

Abstractly, we can say that EasyCrypt targets the automation of a specific verification domain, namely of cryptographic proofs. We note that to this end, the system internally uses first-order automated theorem proving techniques for automating the verification of low-level steps and implements domain-specific reasoning techniques for generating high-level steps. We adopt a similar approach in VeriTaS. However, VeriTaS may be instantiated for multiple verification domains, whereas a system such as EasyCrypt was only ever designed for their target verification domain. Hence, the techniques implemented in this system cannot easily be reused for other verification domains.

9.2.4. Graphical Approaches to Proof Construction

Next, we compare the concept of proof graphs upon which our verification infrastructure is based to other graphical approaches for proof construction. We got the inspiration for proof graphs from the concept of *proof planning* by Richardson and Bundy [RB99]. Notably, in their work, Richardson and Bundy distinguish between the *meta-level logic*, i.e. the (possibly heuristic) logic used for constructing a proof plan, and the *object-level logic*, i.e. the formal system in which the actual proof is constructed. Reasoning at the meta level does not need to be sound, but reasoning at the object level needs to be sound. This corresponds to our requirement of decoupling proof construction and step verification.

The work of Grov et al. [GKL13; GL18] describes a fully graphical implementation of the idea of *proof plans*, which can be used with various existing theorem provers and comes with a GUI for visualizing collections of tactics, called Tinker Tool. To inspect a concrete proof, one can graphically "execute" such a proof plan by passing in the goal as "token" via the root tactic, and then passing the goal token along the tactic nodes like in a petri-net. A tactic may produce several tokens as output, i.e. different subgoals, which can be inspected via the GUI. If an intermediate tactic is not successful, one cannot continue to execute the proof plan.

However, proof plans consist of tactic nodes and thus are similar to tactic languages: Proof plans get stuck if a tactic gets stuck on the way, making it impossible to inspect the remaining hypothetical proof beyond the failure. In contrast, we base our proof graphs on intermediate proof obligations as nodes, which may be constructed as well as inspected even if a proof step further up in the proof graph is not verifiable.

9.2.5. Lemma Generation

The different works by Claessen, Johansson, Rosén, and Smallbone focus on the automated generation of properties about functional programs and on the automated verification of such properties. Firstly, there is QuickSpec [Sma+17], a theory exploration system for automatically discovering equational properties in Haskell programs. QuickSpec uses counterexample generators like QuickCheck in the background to discard false lemmas. QuickSpec is able to generate a number of known laws about functional data structures automatically. HipSpec [Cla+12] attempts additionally to automatically prove the equational properties discovered by QuickSpec by applying induction and translating the resulting steps to first-order logic. Hipster [Joh+14] integrates lemma generation into Isabelle via HipSpec.

HipSpec's methods for automatically proving discovered lemmas are very similar to the methods we use within VeriTaS for automation, especially concerning the translation of proof steps to first-order logic. However, HipSpec targets proving equational laws automatically, while we target more complex properties such as progress and preservation. QuickSpec or Hipster could in principle be used to complement the contributions of this thesis by discovering additional, relevant laws on input specifications. Note however that QuickSpec and Hipster are unable to generate lemmas like progress and preservation properties: Such lemmas require premises, and are thus no equational laws.

9.3. Encoding of proof problems and axiom selection

We compare our work on comparing different encoding strategies for proof problems from Chapter 8 to

1. systems which also encode proof problems to first-order logic and/or employ tools for first-order logic for solving them and could hence benefit from the

results of our empirical study (we mentioned some of these systems before, but discuss them again with regard to our work from Chapter 8),

2. other studies which compare different compilation strategies to first-order logic against each other with regard to prover performance, and to

3. other studies which compare different strategies for axiom selection on problems compiled to first-order logic against each other with regard to prover performance.

9.3.1. Encoding problems in first-order logic

There are a number of general-purpose tools and proof assistants which translate proof problems to first-order logic and apply automated theorem provers on them. We discuss a selection of them, focusing on comparing the concrete encoding strategies that these systems use to our strategies:

The intermediate verification language Boogie 2 [KR10; K. 08] translates problems into the SMT-lib [BST10] format understood by SMT solvers such as Z3 [DB08]. Dafny [Lei10] is a programming language and an automatic program verifier which uses SMT solvers through Boogie 2. Dafny also supports functions and algebraic datatypes, but does not encode function inversion axioms or domain axioms for data types, since such axioms "give rise to enormously expensive disjunctions" [Lei10]. In our study in Chapter 8, we did not observe problems in prover performance with such axioms. Notably, omitting the inversion axioms for functions during axiom selection did not significantly influence prover performance. However, it would be interesting to study the effects of such axioms on prover performance for larger specifications.

Sledgehammer [BP16] is a tool for automating proof steps within the interactive theorem prover Isabelle [Wen12] using automated theorem provers as well as SMT solvers. Sledgehammer encodes general higher-order problems from Isabelle/HOL to first-order logic and SMT-lib. The concrete encodings are described in detail in [MP08; Bla12]. Our encodings differ from the ones that Sledgehammer uses mostly in the details whose effect we study in this article: handling of variable encoding and simplification strategies. Additionally, like Dafny, Sledgehammer does not explicitly encode function inversion or domain axioms.

The higher-order resolution-based theorem prover Leo-II [Ben$^+$15] cooperates with automated first-order theorem provers such as the ones we used by encoding higher-order clauses to first-order clauses. The aforementioned HipSpec [Cla$^+$13] internally compiles definitions and properties from Haskell programs to first-order logic and applies ATPs on them.

All of these tools could benefit from the results of our study for improving their translations to first-order logic or for re-evaluating detailed design decisions within their encoding processes. We believe that our results regarding the encoding of variables may be particularly useful and merit further study: For example, both Dafny and Sledgehammer often introduce auxiliary variables into the first-order compilation to bind subformulas which are used multiple times in the specification.

Our results indicate that inlining such variables may increase prover success rates, at least in certain cases. It would be interesting to further study for which cases our observation about variable inlining applies in more general settings. For example, one could suspect that inlining variables is indeed beneficial for smaller problem specifications, but not for large ones.

Regarding the search for counterexamples, the Alloy Analyzer [Jac06] is a solver that takes constraints of a specification of a model in the Alloy language and tries to find sample structures or counterexamples for these constraints. To achieve this, the Alloy Analyzer reduces a problem to SAT (satisfiability checking) by encoding it to first-order relational logic, which combines elements from first-order logic and relational calculi [Jac00]. Nitpick [BN10b] applies the Alloy Analyzer for finding counterexamples for Isabelle/HOL theorems. The Alloy Analyzer and Nitpick both use the relational model finder Kodkod [TJ07]. In contrast, we investigated using automated first-order theorem provers for exploring whether counterexamples exist. It would be interesting to compare the performance of automated first-order provers for detecting the existence of counterexamples against tools such as Nitpick on a larger set of counterexample goals.

9.3.2. Comparing different compilation strategies

Leino and Rümmer [KR10] empirically compare two different variants of how to translate Boogie 2 types into SMT-lib. They also observed that type guards significantly lower the performance of SMT solvers. Meng and Paulson [MP08] and Blanchette et al. [Bla$^+$13b; Bla12] also investigate different encodings of sorts for Sledgehammer, notably different variations of partial type erasure. Our type erasure encoding and our guard encoding is similar to their encoding variants, but slightly adapts them to our domain. In their studies, the authors of the cited papers also observe that full type guards decrease prover performance, a result which we empirically confirm in our work. Additionally, Meng and Paulson [MP08] and Blanchette [Bla12] also compare different encodings of lambda abstractions against each other, which is outside of the focus of our study, since we deliberately chose benchmarks that avoid lambda abstractions.

Kotelnikov et al. [Kot$^+$16b; Kot$^+$16a] investigate the encoding of a number of constructs which typically occur in specification constructs of language semantics directly within the Vampire theorem prover. Concretely, they adapt the internal input language and calculi of Vampire to support first-class Boolean sorts, let-bindings, and if-then-else expressions. They compare the performance of their encoding strategies with the pure first-order encoding used by Vampire and observe that their encoding increases prover performance for problems which use such constructs. In contrast, we investigate many different compilation strategies for language specifications systematically against each other, including, but not limited to, let-bindings and if-then-else expressions. Another main difference between our work and the one of Kotelnikov et al. is that we treat first-order theorem provers as “black boxes”, while they aim at increasing prover performance by changing the provers internally. The two methods are likely to be complementary.

9.3.3. Comparing different strategies for axiom selection

Meng and Paulson [MP09] investigate axiom selection for problems encoded by Sledgehammer. Since Sledgehammer encodes parts of Isabelle proof problems from many different domains, the encoding typically includes a vast amount of axioms from the standard Isabelle/HOL library. This includes for example a large number of definitions and facts on the built-in list and set constructs. This typically yields very large proof problems for ATPs, where axiom selection seems to be much more important than for our benchmark specifications.

While we investigated domain-specific strategies for axiom selection that were tailored to our specific problems, Meng and Paulson iteratively apply general-purpose heuristics to compute the *relevance mark* of a clause with regard to clauses selected as relevant in a previous iteration, starting with conjecture clauses. Their heuristics include for example the number of functions that appear within a clause, the rarity of a function in the overall clause set, and favoring short clauses. Meng and Paulson empirically compare their strategy for axiom selection to raw problems with regard to prover performance. They find that their selection strategy almost always increases the success rates of the provers.

However, in the conclusion of their paper, Meng and Paulson admit that tuning certain internal weighting mechanisms within ATPs may have just the same effect as their selection strategy. Since the publication of their paper, several modifications such as the SInE strategy [HV11] were added to different ATPs. Hence, it is likely that repeating their empirical study with today's ATPs would yield results more similar to our results from Section 8.4.

This assessment is supported by recent work of Kuksa and Mossakowski [KM16b]: They present an empirical evaluation of a prover-independent and generalized implementation of the SInE selection heuristics and show that this selection strategy alone improves prover performance a lot on problems from Ontohub [KM16a], an open source repository for managing distributed logical theories that focuses on ontologies. Kuksa and Mossakowski also experimented with an extension of SInE, but found that this extension could not improve prover performance further. In comparison, we focused our study on problems arising from exploring language specifications, investigating domain-specific selection strategies. However, from an abstract point of view, we reached a similar conclusion, namely that our domain-specific selection strategies could not improve further on the strategies that are already implemented within ATPs.

Other more recent approaches investigate axiom selection on general proof problems as machine-learning problem: MaSh [Küh+13] implements machine learning for Sledgehammer, learning from a set of given proofs which axioms could be relevant to unseen similar problems. Kühlwein and Blanchette show that MaSh greatly improves prover performance with regard to the earlier selection strategy of Meng and Paulson from above.

Chapter

10

Conclusion and Outlook

10.1. Conclusions

The main goal of this thesis was a solution for obtaining machine-checked type soundness proofs of DSLs with as low an effort as currently possible. Additionally, we were striving for a solution that may, at least in parts, be reused for verification domains beyond our target verification domain of type soundness proofs of DSLs. We were looking for this solution from the perspective of what we call a "domain expert": someone who knows all about conducting proofs in a certain verification domain, but is not necessarily well-versed in using advanced features of current existing theorem provers.

From studying existing works on the mechanizations of type soundness proofs, notably within the context of the PoplMark challenge, we first narrowed down the set of DSLs for which automated type soundness proofs seem feasible today: DSLs without first-class binders, thus circumventing the "name-binding problem" (Section 3.1).

Next, we conducted our own mechanizations of type soundness proofs via progress and preservation for an example type system from such a DSL (remainder of Chapter 3). Via these mechanizations, we analyzed how well existing verification systems are suited for the purposes of our main goal. We concluded that existing systems are not well-suited as a basis for automating domain-specific verification tasks from the perspective of domain experts, as they do not capture the domain-specific concepts within verification domains.

Based on our analysis from Chapter 3, we developed requirements for a verification infrastructure that meets the needs of domain experts in different verification domains. We presented the design and prototypical implementation of a generic verification infrastructure called VeriTaS (Chapter 4). VeriTaS is deliberately designed as a light-weight infrastructure to provide a low entry point for domain experts who want to automate a certain verification domain: The infrastructure is designed as a library within a widely known general-purpose programming language

(Scala), which allows for integrating it easily with other existing infrastructure. VeriTaS is generic in an input format for problem specifications, so that domain experts may easily attach their own, domain-specific input formats.

We proposed a model of proof graphs (Section 4.2), which VeriTaS uses for hierarchically structuring low-level proof problems to an overall proof. Proof graphs in VeriTaS allow for decoupling the generation of a high-level proof structure from the actual verification of the low-level proof steps within the proof. This enables domain experts to generate "approximate" proof structures that may contain incomplete or even incorrect steps. Hence, they can focus on capturing the domain-specific aspects of their proofs when implementing automated proof strategies and do not have to deal with low-level verification details. For verifying low-level proof steps, VeriTaS allows for connecting different ATPs and SMT solvers and makes sure that proof graphs are only considered as "fully verified" once all low-level proof steps have been verified by connected provers. Thus, we ensure the overall correctness of proofs in the end.

We presented a concrete instantiation of our VeriTaS verification infrastructure for the verification domain that this thesis targets: type soundness proofs of DSLs (Chapter 5), thereby also demonstrating the steps needed to instantiate VeriTaS for a particular verification domain. We provided a suitable input format, basic tactics for creating proof steps within proof graphs, and encoding strategies for connecting ATPs and SMT solvers. On top of this instantiation, we developed automated proof strategies that generate proof graphs for progress and preservation proofs of DSLs (Chapter 6).

We demonstrated that our automated strategies are indeed able to successfully generate large parts of progress and preservation proofs for DSLs automatically via two representative case studies (Chapter 7) and discussed the shortcomings of our approach as well as its benefits over existing systems: Firstly, our strategies require users to provide suitable auxiliary lemmas for the proofs. Secondly, users have to enhance their specifications of type systems with domain-specific annotations that guide the top-level strategies and link the given auxiliary lemmas to the correct function specifications. Even then, a small number of proof problems remain within the generated proof graphs that users will have to inspect manually.

However, the auxiliary lemmas needed have a fairly common structure (which we also outlined in this thesis) so that one may assume that the formulation of such properties does not require a lot of effort from a domain expert. Also, the generated proof problems may be inspected within the original specification format so that domain experts do not need to directly work within external provers. Despite the manual effort that our proof strategies still require, we were able to report that our overall approach requires significantly less effort and verification skills by end users than existing verification systems (in particular, Isabelle/HOL and Dafny, which we studied in Chapter 3).

Finally, we evaluated an aspect that turned out to be crucial during the development of VeriTaS: How to encode a proof problem within the input format of existing ATPs (FOL) so that the success rate of these provers is maximized? We conducted an empirical study that compares encoding variants against each other with regard

to the success rate of provers on proof problems within the example specifications we used in this thesis (Chapter 8). Our study provides guidelines to future domain experts for encoding problems to first-order logic.

Final statement Overall, the contributions in this thesis advance the state of the art in two different directions: Firstly, our concrete instantiation of VeriTaS for our target verification domain makes it easier to add soundness proofs to type systems of DSLs. This contribution has the potential of raising the number of reliably sound type systems developed for DSLs. Secondly, with our VeriTaS verification infrastructure as a whole, we suggest a shift of perspective from tools and methods developed for general-purpose verification to tools and methods for domain-specific verification. The ideas and concepts within the design of VeriTaS have the potential of influencing the development of larger, existing verification systems towards more support for developing domain-specific formats and verification strategies.

10.2. Future Directions of Research

This thesis opens up several interesting directions for future research. In the following, we discuss potential extensions of the work presented here and how the concepts from this work may be integrated with other existing works. The first two subsections focus on possible future work on our instantiation of VeriTaS for type soundness proofs. The last subsection discusses potential future work in the general area of domain-specific verification, based on the concepts proposed within this thesis.

10.2.1. Enlarging the Language Focus

In this thesis, we focused on automating type soundness proofs of DSLs without first-class binders and without subtyping. A future extension of our instantiation of VeriTaS could extend the available input formats for specifications of languages and type systems as well as our proof strategies to support more languages. Such languages may include other, more complex DSLs — or even general-purpose languages, since some DSLs essentially are as expressive as general-purpose languages. (For example, PostScript, a page description language used within printers, is in principle usable for any programming domain.)

Subtyping The most straightforward extension to enlarge the language focus of this thesis is to add support for languages with subtyping. For this, one could firstly add syntactic sugar to ScalaSPL for subtyping, similar to the existing syntactic sugar for typing judgments. As a next step, one would need to analyze reasoning patterns in progress and preservation proofs for languages with subtyping. These patterns include for example induction on typing derivations, which is needed as a top-level step for progress and preservation proofs for type systems with a classical subsumption rule (which enables typing a single term with one or more subtypes of its actual type). Furthermore, techniques for proving standard properties such as

transitivity of subtyping will need to be analyzed and automated. The associated problems are not trivial, as the PoplMark challenge also showed (one part of the challenge was to mechanize a transitivity proof for standard subtyping).

First-class binders Supporting first-class binders in machine-checked proofs is a long-standing research problem. Some solutions to this problem exist, all with different advantages and inconveniences. We discussed the problem for supporting first-class binders and the existing solutions in Section 3.1.1. Our proof strategies for type soundness proofs currently do not attempt at all to generate sensible proof steps for languages with first-class binders, since we excluded such languages from our focus.

Note, however, that one could nevertheless attempt to use our instantiation of VeriTaS to conduct type soundness proofs for languages with first-class binders: One would have to implement a custom substitution function for instantiating binders and use it within a top-level reduction function. That is, the substitution function would be like any other auxiliary function. One could state the classical substitution property ("substituting a name does not change the type of the expression") as a preservation property. Our proof strategies would then, as expected, create lemma application steps with the substitution property at the appropriate places within a type soundness proof. For proving the substitution property, our strategies would generate induction and case distinction steps as induced by the structure of the substitution function. This alone, however, would not suffice for proving the substitution property: The proof would be stuck in the cases where an actual substitution took place. Verifying this step would require a lot of manual interactions with the generated proof graph.

One could attempt to automate one of the approaches for mechanized reasoning with first-class binders mentioned in Section 3.1.1. This automation could be implemented within refined proof strategies for type soundness proofs. The most promising approach here might be to implement and use a variant of nominal logic: One could encode axioms on alpha-equivalence of terms and include them when encoding proof problems. Furthermore, one would need to work out common reasoning patterns for proof steps that require these axioms and encode them within the proof strategies.

Improving the input format for language specifications ScalaSPL as presented in this thesis consists of core constructs for language specifications. While these basic constructs theoretically allow for modeling arbitrary language specifications, the modeling of some language constructs may be rather cumbersome. An example for this is the modeling necessary for tables for our typed SQL case study (see Chapter 7), which required defining different list constructs from scratch. By adding more features to ScalaSPL to support language specifications, such as for example structures from the Scala library and type parameters, we may indirectly enlarge the language focus of our instantiation of VeriTaS: A more expressive specification language allows for modeling new languages faster.

10.2.2. Raising the Degree of Automation Further

Either together or independently of enlarging the language focus of our instantiation of VeriTaS, we may extend and refine the existing proof strategies to raise the overall degree of automation even further.

Refining proof strategies We implemented basic proof strategies for generating simple progress and preservation proofs automatically and demonstrated via two case studies that these strategies are well-suited for automating large parts of such proofs. However, there is of course a lot of potential for refining the basic proof strategies that we presented in this thesis. Firstly, as we already hinted at when describing the strategies that generate lemma application steps, one may implement refined lemma selection strategies that select only lemmas that are likely to be useful for the present proof step. This would require further inspection of the language specification and/or more user annotations. Secondly, our basic strategies currently do not interact with verifiers at all. One could implement proof strategies that themselves attempt the verification of a proof step and then apply a different strategy if the verification is inconclusive. However, the "price to pay" for such strategies would be that the generation of a proof graph might take longer.

Lemma generation Our automated proof strategies for type soundness proofs of DSLs currently require users to specify auxiliary lemmas that have a certain structure. By implementing strategies that generate as many of such lemmas automatically as possible, one could raise the degree of automation achieved in this thesis again considerably. Such a lemma generation could for example proceed by generating relevant combinations of static and dynamic conditions, according to the patterns for progress and preservation properties that we described. A strategy may obtain these conditions from inspecting the domain-specific knowledge available for specifications (`@Static` and `@Dynamic` annotations of functions).

The author of this thesis supervised the master's thesis of Weber [Web19], who explored approaches for lemma generation for the case study of typed SQL from this thesis. Weber was able to successfully generate a large majority of the auxiliary lemmas that this case study requires, while at the same time keeping the overall number of generated lemmas to a feasible number that a user might still inspect (below 100).

Weber presented 3 different algorithms to explore different directions for lemma generation: His first approach naively combined all possible static and dynamic conditions from a language specification according to progress and preservation patterns. For the generation of concrete premises and conclusions, Weber firstly exploits the domain-specific annotations we defined for specifications of type systems in ScalaSPL in Section 6.2. Secondly, Weber used the extensible annotation system of ScalaSPL to add further light-weight domain-specific annotations. For example, Weber uses an annotation named `@Failable` to mark functions in the specification that may fail. This allows him to decide for which functions he attempts the generation of a progress property.

Weber's first algorithm produced the necessary auxiliary lemmas, but also thousands of "nonsense" lemmas. Therefore, in his second algorithm, Weber generates lemmas in a more structured fashion and applies the verifiers we connected to VeriTaS to detect false lemmas and delete them from the set of lemmas presented at the end. Furthermore, the set of lemmas presented to an end user may be manipulated via a selection strategy. This second algorithm produced far less lemmas than Weber's first approach, but was not able to generate some necessary lemmas. In a third algorithm, Weber added further domain-specific annotations that allow for giving "hints" to the lemma generation algorithm. This approach allowed for generating the majority of lemmas required for the typed SQL case study.

Weber's work confirmed that it is possible to use light-weight domain-specific annotations in ScalaSPL to successfully guide a lemma generation strategy. Especially, Weber's work confirmed that it is possible to add complex domain-specific strategies to VeriTaS without having deep internal knowledge of the system: When Weber started his master's thesis, he was neither familiar with the Scala language, nor with the code base of VeriTaS.

The scope of Weber's thesis only allowed for exploring a single case study and a limited amount of different strategies and generation heuristics. One could extend Weber's work by refining his strategies for lemma generation, exploring the use of different internal heuristics, and trying more case studies. Finally, one could integrate Weber's lemma generation with the proof strategies we presented in this thesis.

10.2.3. Domain-specific Verification

We designed and implemented the VeriTaS verification infrastructure according to a number of general concepts that we identified as important for supporting domain experts with automation tasks in their verification domain. We discuss how our concepts and results could be used in other verification domains and how they could influence improvements in the area of verification in general.

Other verification domains Other examples of verification domains that could be automated within VeriTaS could be *noninterference* proofs from the area of information-flow security, *reduction proofs* from the area of cryptography, or termination proofs for programs. All of these proofs have in common that they typically have a certain recurring overall proof structure and that certain high-level techniques keep being reused. For example, noninterference proofs typically argue about *bisimulations* and apply *unwinding techniques*. Reduction proofs in cryptography typically aim at reducing the problem of attacking a cryptographic protocol to certain recurring hard-to-solve problems from cryptography. Termination proofs typically use a *measure term* and prove that this term decreases over the course of a program.

VeriTaS enables domain experts in such verification domains to add their own format and abstractions for such high-level proof techniques. In particular, VeriTaS does not enforce a specific strategy for proof generation. We believe that

these features are crucial to get more domain experts to attempt automating the mechanization of proofs in their domain.

Better support for domain-specific verification in existing systems Existing general-purpose verification systems such as Isabelle and Coq could adopt parts of our design for domain-specific verification in order to better support domain experts. For example, one could add generic fontends to existing systems similar to our VeriTaS verification infrastructure, with API constructs that allow for connecting to existing verification systems. Such a generic frontend could offer support for attaching and translating a domain-specific input format to the general-purpose format of the verification infrastructure. Furthermore, such a frontend could provide light-weight API methods for automatically generating proof structures in existing systems using domain-specific methods and formats. And finally, such a frontend could provide support for manipulating the output of proofs from existing systems, in order to translate the output to a human-readable, domain-specific format.

Coming from the other side, i.e. the side of domain experts, it would have an equivalent effect to better integrate support for automated verification into existing language workbenches such as Spoofax [KV10; Vis+14]. To this end, the design principles from our VeriTaS verification infrastructure could also be employed within existing language workbenches, connecting them to existing verifiers and enabling the implementation of domain-specific verification strategies from within a language workbench.

Improving the usage of existing ATPs The empirical comparison study on encoding strategies that we presented in Chapter 8 of this thesis helped us to optimize our encoding of proof problems arising within type soundness proofs of DSLs.

Beyond this concrete purpose, our study provides general guidance regarding how to encode a domain-specific proof problem for existing ATPs. Furthermore, on a meta-level, our empirical comparison study can be seen as one of many instances of a study for fine-tuning domain-specific encoding strategies. One could port the overall setup of our study to other verification domains in order to systematically compare encoding alternatives with regard to the success rates of ATPs. The results of such comparison studies may help to raise the overall degree of automation in different verification domains.

Last but not least, one may use such studies to systematically generate more proof problems for the TPTP [Sut17], which will ultimately serve to improve the overall performance of existing ATPs who use the TPTP as benchmarks. The problems generated within this thesis were also submitted to the TPTP for this purpose.

Bibliography

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016.

[All70] Frances E. Allen. "Control Flow Analysis." In: *SIGPLAN Not.* 5(7), July 1970: pp. 1–19.

[ASS08] Jonathan Aldrich, Robert J. Simmons, and Key Shin. "SASyLF: An Educational Proof Assistant for Language Theory." In: *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*. ACM, 2008, pp. 31–40.

[AW13] Noran Azmy and Christoph Weidenbach. "Computing Tiny Clause Normal Forms." In: *CADE*. 2013, pp. 109–125.

[Ayd+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. "Mechanized Metatheory for the Masses: The PoplMark Challenge." In: *Proceedings of International Conference on Theorem Proving in Higher Order Logics (TPHOL)*. Springer-Verlag, 2005, pp. 50–65.

[Bal04] Clemens Ballarin. "Locales and Locale Expressions in Isabelle/Isar." In: *Types for Proofs and Programs*. Springer Berlin Heidelberg, 2004, pp. 34–50.

[Bar+11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer." In: *Proceedings of Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*. 2011, pp. 71–90.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers." In: *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*. 2011, pp. 116–130.

[Ben+15] Christoph Benzmüller, Lawrence C. Paulson, Nik Sultana, and Frank Theiß. "The Higher-Order Prover LEO-II." In: *Journal of Automated Reasoning*, 2015.

[Ber07] Stefan Berghofer. "POPLmark Challenge Via de Bruijn Indices." In: *Archive of Formal Proofs* 2007, 2007.

[Ber12] Stefan Berghofer. "A Solution to the PoplMark Challenge Using de Bruijn Indices in Isabelle/HOL." In: *J. Autom. Reasoning* 49(3), 2012: pp. 303–326.

[Bet+12] Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. "Approaches and Tools for Implementing Type Systems in Xtext." In: *Software Language Engineering, 5th International Conference, SLE Revised Selected Papers*. 2012, pp. 392–412.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[Bla12] Jasmin C. Blanchette. "Automatic Proofs and Refutations for Higher-order Logic." PhD thesis. Technical University Munich, 2012.

[Bla+13a] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. "An Overview of the Leon Verification System: Verification by Translation to Recursive Functions." In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. ACM, 2013.

[Bla+13b] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. "Encoding Monomorphic and Polymorphic Types." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013, pp. 493–507.

[BN10a] Jasmin Christian Blanchette and Tobias Nipkow. "Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder." In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. 2010, pp. 131–146.

[BN10b] Jasmin Christian Blanchette and Tobias Nipkow. "Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder." In: *Proceedings of International Conference on Interactive Theorem Proving (ITP)*. 2010, pp. 131–146.

[Bob+11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Why3: Shepherd Your Herd of Provers." In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64.

[BP16] Jasmin C. Blanchette and Lawrence C. Paulson. *Hammering Away - A User's Guide to Sledgehammer for Isabelle/HOL*. Tech. rep. 2016.

[Bra+18] Oliver Bracevac, Richard Gay, Sylvia Grewe, Heiko Mantel, Henning Sudbrock, and Markus Tasch. "An Isabelle/HOL Formalization of the Modular Assembly Kit for Security Properties." In: *Archive of Formal Proofs* 2018, 2018.

[Bru72] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." In: *Indagationes Mathematicae (Proceedings)* 75(5), 1972: pp. 381 –392.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0." In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[Bul12] Lukas Bulwahn. "The New Quickcheck for Isabelle." In: *Certified Programs and Proofs*. Springer Berlin Heidelberg, 2012, pp. 92–108.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '74. 1974.

[Cha12] Arthur Charguéraud. "The Locally Nameless Representation." In: *J. Autom. Reasoning* 49(3), 2012: pp. 363–408.

[Cla+12] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. "HipSpec: Automating Inductive Proofs of Program Properties." In: *ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation, Manchester, UK, June 2012*. 2012, pp. 16–25.

[Cla+13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. "Automating Inductive Proofs Using Theory Exploration." In: *Proceedings of International Conference on Automated Deduction (CADE)*. 2013, pp. 392–406.

[CLS11] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. "Sort It out with Monotonicity: Translating Between Many-sorted and Unsorted First-order Logic." In: *Proceedings of International Conference on Automated Deduction (CADE)*. Springer, 2011, pp. 207–221.

[CMS16] Matteo Cimini, Dale Miller, and Jeremy G. Siek. "Well-Typed Languages are Sound." In: *CoRR* abs/1611.05105, 2016.

[Con+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. NJ: Prentice-Hall, 1986.

[CS12] Alberto Ciaffaglione and Ivan Scagnetto. "A weak HOAS approach to the POPLmark Challenge." In: *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012, Rio de Janeiro, Brazil, September 29-30, 2012*. 2012, pp. 109–124.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[DD77] Dorothy E. Denning and Peter J. Denning. "Certification of Programs for Secure Information Flow." In: *Commun. ACM* 20(7), 1977: pp. 504–513.

[Del00] David Delahaye. “A Tactic Language for the System Coq.” In: *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2000, pp. 85–95.

[Den76] Dorothy E. Denning. “A Lattice Model of Secure Information Flow.” In: *Commun. ACM* 19(5), 1976: pp. 236–243.

[Don18] Jacob Donham. “A domain-specific language for microservices.” In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*. 2018, pp. 2–12.

[EB10] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way.” In: *SPLASH*. 2010, pp. 307–309.

[EFO14] Sebastian Erdweg, Stefan Fehrenbach, and Klaus Ostermann. “Evolution of Software Systems with Extensible Languages and DSLs.” In: *IEEE Software* 31(5), 2014: pp. 68–75.

[EFT94] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer, 1994.

[End+14] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. “How do API documentation and static typing affect API usability?” In: *ICSE*. 2014, pp. 632–642.

[ER13] Sebastian Erdweg and Felix Rieger. “A framework for extensible languages.” In: *Generative Programming: Concepts and Experiences*. 2013, pp. 3–12.

[Erd+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. “The State of the Art in Language Workbenches.” In: *Proceedings of Conference on Software Language Engineering (SLE)*. Vol. 8225. LNCS. Springer, 2013, pp. 197–217.

[Erd+15] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. “Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future.” In: 44, Part A, 2015: pp. 24–47.

[Fow11] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers." In: *Proceedings of the 22nd European Symposium on Programming*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, Mar. 2013, pp. 125–128.

[FV06] J. Nathan Foster and Dimitrios Vytiniotis. "A Theory of Featherweight Java in Isabelle/HOL." In: *Archive of Formal Proofs*, 2006. `http://isa-afp.org/entries/FeatherweightJava.html`, Formal proof development.

[Gac08] Andrew Gacek. "The Abella Interactive Theorem Prover (System Description)." In: *IJCAR*. Springer, 2008, pp. 154–161.

[GEM15] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. "Using Vampire in Soundness Proofs of Type Systems." In: *Proceedings of the 1st and 2nd Vampire Workshops*. EPiC, 2015, pp. 33–51.

[GEM16] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. "Automating Proof Steps of Progress Proofs: Comparing Vampire and Dafny." In: *Proceedings of the 3rd Vampire Workshop*. 2016, pp. 33–45.

[GKL13] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. "A Graphical Language for Proof Strategies." In: *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2013, pp. 324–339.

[GL18] Gudmund Grov and Yuhui Lin. "The Tinker tool for graphical tactic development." In: *STTT* 20(2), 2018: pp. 139–155.

[GMS14] Sylvia Grewe, Heiko Mantel, and Daniel Schoepe. "A Formalization of Assumptions and Guarantees for Compositional Noninterference." In: *Archive of Formal Proofs* 2014, 2014.

[Gou+] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. "Verifying OpenJDK's Sort Method for Generic Collections." In: *Journal of Automated Reasoning* 62(1), pp. 93–126.

[GPM18] Sylvia Grewe, André Pacak, and Mira Mezini. "Using Vampire with Support for Algebraic Datatypes in Type Soundness Proofs." In: *Vampire 2017. Proceedings of the 4th Vampire Workshop*. Ed. by Laura Kovács and Andrei Voronkov. Vol. 53. EPiC Series in Computing. 2018, pp. 42–51.

[Gre16] Sylvia Grewe. "VeriTaS: Verification of Type System Specifications: Mechanizing Domain Knowledge about Progress and Preservation Proofs." In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. 2016, pp. 12–14.

[Gre+14a] Sylvia Grewe, Alexander Lux, Heiko Mantel, and Jens Sauer. "A Formalization of Declassification with WHAT-and-WHERE-Security." In: *Archive of Formal Proofs* 2014, 2014.

[Gre+14b] Sylvia Grewe, Alexander Lux, Heiko Mantel, and Jens Sauer. "A Formalization of Strong Security." In: *Archive of Formal Proofs* 2014, 2014.

[Gre+15] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. "Type Systems for the Masses: Deriving Soundness Proofs and Efficient Checkers." In: *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*. ACM, 2015, pp. 137–150.

[Gre+16] Sylvia Grewe, Sebastian Erdweg, Michael Raulf, and Mira Mezini. "Exploration of Language Specifications by Compilation to First-Order Logic." In: *Proceedings of International Symposium on Principles and Practice of Declarative Programming (PPDP)*. 2016, pp. 104–117.

[Gre+18a] Sylvia Grewe, Sebastian Erdweg, André Pacak, Michael Raulf, and Mira Mezini. "Exploration of Language Specifications by Compilation to First-Order Logic." In: *Sci. Comput. Program.* 155, 2018: pp. 146–172.

[Gre+18b] Sylvia Grewe, Sebastian Erdweg, André Pacak, and Mira Mezini. "System Description: An Infrastructure for Combining Domain Knowledge with Automated Theorem Provers." In: *Proceedings of International Symposium on Principles and Practice of Declarative Programming (PPDP)*. 2018, 24:1–24:10.

[GT16] Gudmund Grov and Vytautas Tumas. "Tactics for the Dafny Program Verifier." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2016, pp. 36–53.

[Han+14] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. "An empirical study on the impact of static typing on software maintainability." In: *Empirical Software Engineering* 19(5), 2014: pp. 1335–1382.

[Hau+16] Markus Hauck, Savvas Savvides, Patrick Eugster, Mira Mezini, and Guido Salvaneschi. "SecureScala: Scala embedding of secure computations." In: *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*. 2016, pp. 75–84.

[HK16] Lars Hupel and Viktor Kuncak. "Translating Scala Programs to Isabelle/HOL." In: *IJCAR*. 2016, pp. 568–577.

[HL07] Robert Harper and Daniel R. Licata. "Mechanizing Metatheory in a Logical Framework." In: *Functional Programming*, 2007: pp. 613–673.

[HTK00] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. Cambridge, MA, USA: MIT Press, 2000.

[HV11] Krystof Hoder and Andrei Voronkov. "Sine Qua Non for Large Theory Reasoning." In: *Proceedings of International Conference on Automated Deduction (CADE)*. 2011, pp. 299–314.

[IPW01a] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ." In: *ACM Trans. Program. Lang. Syst.* 23(3), 2001.

[IPW01b] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ." In: *ACM Trans. Program. Lang. Syst.* 23(3), 2001: pp. 396–450.

[Jac00] Daniel Jackson. "Automating first-order relational logic." In: *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. 2000, pp. 130–139.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[Joh+14] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. "Hipster: Integrating Theory Exploration in a Proof Assistant." In: *CoRR* abs/1405.3426, 2014.

[JSH13] Reiner Jung, Christian Schneider, and Wilhelm Hasselbring. "Type Systems for Domain-specific Languages." In: *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik*. 2013, pp. 139–154.

[Kle+12a] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. "Run your research: On the effectiveness of lightweight mechanization." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. ACM, 2012, pp. 285–296.

[Kle+12b] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. "Do static type systems improve the maintainability of software systems? An empirical study." In: *IEEE 20th International Conference on Program Comprehension, ICPC*. 2012, pp. 153–162.

[KM16a] Eugen Kuksa and Till Mossakowski. "Ontohub: Version Control, Linked Data and Theorem Proving for Ontologies." In: *Proceedings of the Joint Ontology Workshops 2016 Episode 2*. 2016.

[KM16b] Eugen Kuksa and Till Mossakowski. "Prover-independent Axiom Selection for Automated Theorem Proving in Ontohub." In: *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR)*. 2016, pp. 56–68.

[KN05] Gerwin Klein and Tobias Nipkow. "Jinja is not Java." In: *Archive of Formal Proofs*, 2005. `http://isa-afp.org/entries/Jinja.html`, Formal proof development.

[KN06] Gerwin Klein and Tobias Nipkow. "A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler." In: *ACM Trans. Program. Lang. Syst.* 28(4), 2006: pp. 619–695.

[Kot+16a] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. "A Clausal Normal Form Translation for FOOL." In: *Global Conference on Artificial Intelligence (GCAI)*. Vol. 41. EPiC Series in Computing. 2016, pp. 53–71.

[Kot+16b] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. "The Vampire and the FOOL." In: *Proceedings of the ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*. 2016, pp. 37–48.

[KR10] K. Rustan M. Leino and Philipp Rümmer. "A Polymorphic Intermediate Verification Language: Design and Logical Encoding." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2010.

[KRV17] Laura Kovács, Simon Robillard, and Andrei Voronkov. "Coming to Terms with Quantified Reasoning." In: *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 2017, pp. 260–270.

[KV10] Lennart C.L. Kats and Eelco Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. ACM, 2010, pp. 444–463.

[KV13] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire." In: *Proceedings of International Conference on Computer Aided Verification (CAV)*. Springer, 2013, pp. 1–35.

[Küh+13] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. "MaSh: Machine Learning for Sledgehammer." In: *Proceedings of International Conference on Interactive Theorem Proving (ITP)*. 2013, pp. 35–50.

[LCH06] Daniel K. Lee, Karl Crary, and Robert Harper. "Mechanizing the Metatheory of Standard ML." In: 2006.

[LE13] Florian Lorenzen and Sebastian Erdweg. "Modular and Automated Type-Soundness Verification for Language Extensions." In: *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, 2013, pp. 331–342.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness." In: *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2010, pp. 348–370.

[Ler07] Xavier Leroy. *A locally nameless solution to the POPLmark challenge*. Research Report. 2007, p. 54.

[LR91] William Landi and Barbara G. Ryder. "Pointer-induced Aliasing: A Problem Classification." In: *POPL*. 1991, pp. 93–103.

[LR92] William Landi and Barbara G. Ryder. "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing." In: *PLDI*. 1992, pp. 235–248.

[MMW16] Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. "Eisbach: A Proof Method Language for Isabelle." In: *J. Autom. Reasoning* 56(3), 2016: pp. 261–282.

[MP08] Jia Meng and Lawrence C. Paulson. "Translating Higher-Order Clauses to First-Order Clauses." In: *J. Autom. Reasoning* 40(1), 2008: pp. 35–60.

[MP09] Jia Meng and Lawrence C. Paulson. "Lightweight relevance filtering for machine-generated resolution problems." In: *Journal of Applied Logic*, 2009: pp. 41–57.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[OSV11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, 2011.

[Pac18] André Pacak. "Designing and Evaluating an Input Format for Generating Basic Type Soundness Proofs via Domain-Specific Proof Strategies." MA thesis. TU Darmstadt, 2018.

[Pap+11] Stergios Papadimitriou, Constantinos Terzidis, Seferina Mavroudi, and Spiridon D. Likothanassis. "ScalaLab: An Effective Scala-Based Scientific Programming Environment for Java." In: *Computing in Science and Engineering* 13(5), 2011: pp. 43–55.

[Pfe91] Frank Pfenning. "Logical Frameworks." In: ed. by Gérard Huet and Gordon Plotkin. Cambridge University Press, 1991. Chap. Logic Programming in the LF Logical Framework, pp. 149–181.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.

[Pit03] Andrew M. Pitts. "Nominal logic, a first order theory of names and binding." In: *Information and Computation* 186(2), 2003: pp. 165 –193.

[PS99] Frank Pfenning and Carsten Schürmann. "System Description: Twelf - A Meta-Logical Framework for Deductive Systems." In: *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*. 1999, pp. 202–206.

[RB99] Julian Richardson and Alan Bundy. "Proof Planning Methods as Schemas." In: *J. Symbolic Computation* 11, 1999: pp. 1–000.

[Rob+08] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. "Efficient Software Model Checking of Soundness of Type Systems." In: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2008, pp. 493–504.

[RSV16] Giles Reger, Martin Suda, and Andrei Voronkov. "New Techniques in Clausal Form Generation." In: *GCAI 2016. 2nd Global Conference on Artificial Intelligence, September 19 - October 2, 2016, Berlin, Germany*. 2016, pp. 11–23.

[RV01] Alan Robinson and Andrei Voronkov, eds. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., 2001.

[Ryd79] Barbara G. Ryder. "Constructing the Call Graph of a Program." In: *IEEE Trans. Softw. Eng.* 5(3), May 1979: pp. 216–226.

[Rüm08a] Philipp Rümmer. "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 2008, pp. 274–289.

[Rüm08b] Philipp Rümmer. "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic." In: *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer, 2008, pp. 274–289.

[Rüm18] Philipp Rümmer. *Princess theorem prover*. `http://www.philipp.ruemmer.org/princess.shtml`. 2018.

[Sch13] Stephan Schulz. "System Description: E 1.8." In: *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Vol. 8312. LNCS. Springer, 2013, pp. 735–743.

[Sew+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. "Ott: Effective tool support for the working semanticist." In: *Functional Programming* 20(1), 2010: pp. 71–122.

[SG02] Don Syme and Andrew D. Gordon. "Automating Type Soundness Proofs via Decision Procedures and Guided Reductions." In: *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer, 2002, pp. 418–434.

[Sma+17] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. "Quick specifications for the busy programmer." In: *J. Funct. Program.* 27, 2017: e18.

[SP98] Carsten Schürmann and Frank Pfenning. "Automated Theorem Proving in a Simple Meta-Logic for LF." In: *Proceedings of the 15th International Conference on Automated Deduction: Automated Deduction*. Proceedings of International Conference on Automated Deduction (CADE). 1998, pp. 286–300.

[Sut10] Geoff Sutcliffe. "The TPTP World - Infrastructure for Automated Reasoning." In: *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer-Verlag, 2010, pp. 1–12.

[Sut17] Geoff Sutcliffe. "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0." In: *Journal of Automated Reasoning* 59(4), 2017: pp. 483–502.

[TJ07] Emina Torlak and Daniel Jackson. "Kodkod: A Relational Model Finder." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2007, pp. 632–647.

[UT05] Christian Urban and Christine Tasson. "Nominal Techniques in Isabelle/HOL." In: *Proceedings of International Conference on Automated Deduction (CADE)*. 2005, pp. 38–53.

[Vis+14] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vald Vergu, Augusto Passalaqua, and Gabriël Konat. "A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs." In: *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*. ACM, 2014, pp. 95–111.

[Vou12] Jérôme Vouillon. "A Solution to the PoplMark Challenge Based on de Bruijn Indices." In: *J. Autom. Reasoning* 49(3), 2012: pp. 327–362.

[VP12] Markus Voelter and Vaclav Pech. "Language modularity with the MPS language workbench." In: *ICSE*. 2012, pp. 1449–1450.

[Web19] Friedrich Weber. "Automated Lemma Generation for Soundness Proofs of Type Systems for DSLs." MA thesis. TU Darmstadt, 2019.

[Wen02] Markus Wenzel. "Isabelle, Isar - A Versatile Environment for Human-Readable Formal Proof Documents." PhD thesis. Technical University Munich, Germany, 2002.

[Wen12] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. 2012.

[WF94] Andrew K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness." In: *Inf. Comput.* 115(1), 1994: pp. 38–94.

[Zil+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. "Mtac: A Monad for Typed Tactic Programming in Coq." In: *Proceedings of International Conference on Functional Programming (ICFP)*. 2013, pp. 87–100.

[Isa18] Isabelle Development Team. *Isabelle documentation*. `http://isabelle.in.tum.de/documentation.html`. 2018.

[K. 08] K. Rustan M. Leino. *This is Boogie 2*. Tech. rep. 2008.

[Tea19] The Coq Development Team. *The Coq Reference Manual, version 8.9.0*. Available electronically at `http://coq.inria.fr/doc`. 2019.

Appendix

A

Full Specifications in ScalaSPL

A.1. Typed Arithmetic Expressions

```
1  package de.tu_darmstadt.veritas.scalaspl
2
3  import de.tu_darmstadt.veritas.scalaspl.lang.ScalaSPLSpecification
4
5
6  // specification of typed arithmetic expressions as given in Pierce, TAPL, Chapters 3 and 8
7  // added a Plus operation
8  object AESpec extends ScalaSPLSpecification {
9
10   //simple Boolean and arithmetic expressions
11   sealed trait Term extends Expression
12
13   case class True() extends Term
14
15   case class False() extends Term
16
17   case class Ifelse(b: Term, t: Term, e: Term) extends Term
18
19   case class Zero() extends Term
20
21   case class Succ(p: Term) extends Term
22
23   case class Pred(s: Term) extends Term
24
25   case class Iszero(t: Term) extends Term
26
27   case class Plus(t1: Term, t2: Term) extends Term
28
29   @Dynamic
30   @Recursive(0)
```

```
31  @AuxiliaryProperty("isNVisNat")
32  def isNV(t: Term): Boolean = t match {
33    case Zero() => true
34    case Succ(nv) => isNV(nv)
35    case _ => false
36  }
37
38  def isValue(t: Term): Boolean = t match {
39    case True() => true
40    case False() => true
41    case t1 => isNV(t1)
42  }
43
44  @FailableType
45  sealed trait OptTerm
46
47  case class noTerm() extends OptTerm
48
49  case class someTerm(t: Term) extends OptTerm
50
51  def isSomeTerm(t: OptTerm): Boolean = t match {
52    case noTerm() => false
53    case someTerm(_) => true
54  }
55
56  @Partial
57  def getTerm(ot: OptTerm): Term = ot match {
58    case someTerm(t) => t
59  }
60
61  @Dynamic
62  @Recursive(1)
63  @PreservationProperty("PlusPreservation")
64  def plusop(t: Term, t1: Term): Term = (t, t1) match {
65    case (t2, Zero()) => t2
66    case (t2, Succ(t3)) => Succ(plusop(t2, t3))
67    case (t2, t3) => t3
68  }
69
70  //reduction semantics for simple Boolean and arithmetic terms
71  @ProgressProperty("Progress")
72  @PreservationProperty("Preservation")
73  @Recursive(0)
74  @Dynamic
75  def reduce(t: Term): OptTerm = t match {
76    case Ifelse(True(), t2, t3) => someTerm(t2)
77    case Ifelse(False(), t2, t3) => someTerm(t3)
78    case Ifelse(t1, t2, t3) =>
79      val ot1 = reduce(t1)
80      if (isSomeTerm(ot1))
81        someTerm(Ifelse(getTerm(ot1), t2, t3))
```

```
82        else
83          noTerm()
84      case Succ(t1) =>
85        val ot2 = reduce(t1)
86        if (isSomeTerm(ot2))
87          someTerm(Succ(getTerm(ot2)))
88        else
89          noTerm()
90      case Pred(Zero()) => someTerm(Zero())
91      case Pred(Succ(nv)) =>
92        if (isNV(nv))
93          someTerm(nv)
94        else {
95          val ot2 = reduce(Succ(nv))
96          if (isSomeTerm(ot2))
97            someTerm(Pred(getTerm(ot2)))
98          else
99            noTerm()
100       }
101     case Pred(t1) =>
102       val ot2 = reduce(t1)
103       if (isSomeTerm(ot2))
104         someTerm(Pred(getTerm(ot2)))
105       else
106         noTerm()
107     case Iszero(Zero()) => someTerm(True())
108     case Iszero(Succ(nv)) =>
109       if (isNV(nv))
110         someTerm(False())
111       else {
112         val ot2 = reduce(Succ(nv))
113         if (isSomeTerm(ot2))
114           someTerm(Iszero(getTerm(ot2)))
115         else
116           noTerm()
117       }
118     case Iszero(t1) =>
119       val ot2 = reduce(t1)
120       if (isSomeTerm(ot2))
121         someTerm(Iszero(getTerm(ot2)))
122       else
123         noTerm()
124     case Plus(t1, t2) =>
125       if (isNV(t1))
126         if (isNV(t2))
127           someTerm(plusop(t1, t2))
128         else {
129           val ot2 = reduce(t2)
130           if (isSomeTerm(ot2))
131             someTerm(Plus(t1, getTerm(ot2)))
132           else
```

```
133             noTerm()
134         }
135       else {
136         val ot1 = reduce(t1)
137         if (isSomeTerm(ot1))
138           someTerm(Plus(getTerm(ot1), t2))
139         else
140           noTerm()
141       }
142     case _ => noTerm()
143   }
144
145   //types (Bool and Nat)
146   sealed trait Ty extends Type
147
148   case class B() extends Ty
149
150   case class Nat() extends Ty
151
152   //typing rules
153   @Axiom
154   def Ttrue(): Unit = {} ensuring (True() :: B())
155
156   @Axiom
157   def Tfalse(): Unit = {} ensuring (False() :: B())
158
159   @Axiom
160   def Tif(t1: Term, t2: Term, t3: Term, T: Ty): Unit = {
161     require(t1 :: B())
162     require(t2 :: T)
163     require(t3 :: T)
164   } ensuring (Ifelse(t1, t2, t3) :: T)
165
166   //inversion axiom for Ifelse
167   //@Axiom
168   //def Tif_inv(t1: Term, t2: Term, t3: Term, T: Ty): Unit = {
169   // require(Ifelse(t2, t2, t3) :: T)
170   //} ensuring((t1 :: B()) && (t2 :: T) && (t3 :: T))
171
172   //inversion axioms for Ifelse, since the above version causes translation problems:
173   @Axiom
174   def Tif_inv1(t1: Term, t2: Term, t3: Term, T: Ty): Unit = {
175     require(Ifelse(t1, t2, t3) :: T)
176   } ensuring (t1 :: B())
177
178   @Axiom
179   def Tif_inv2(t1: Term, t2: Term, t3: Term, T: Ty): Unit = {
180     require(Ifelse(t1, t2, t3) :: T)
181   } ensuring (t2 :: T)
182
183   @Axiom
```

```
184  def Tif_inv3(t1: Term, t2: Term, t3: Term, T: Ty): Unit = {
185    require(Ifelse(t1, t2, t3) :: T)
186  } ensuring (t3 :: T)
187
188
189  @Axiom
190  def TZero(): Unit = {} ensuring (Zero() :: Nat())
191
192  //inversion axiom for TZero
193  @Axiom
194  def TZero_inv(T: Ty): Unit = {
195    require(Zero() :: T)
196  } ensuring(T == Nat())
197
198  @Axiom
199  def TSucc(t1: Term): Unit = {
200    require(t1 :: Nat())
201  } ensuring (Succ(t1) :: Nat())
202
203  //inversion axioms for TSucc
204  @Axiom
205  def TSucc_inv1(t1: Term, T: Ty): Unit = {
206    require(Succ(t1) :: T)
207  } ensuring (T == Nat())
208
209  @Axiom
210  def TSucc_inv2(t1: Term): Unit = {
211    require(Succ(t1) :: Nat())
212  } ensuring (t1 :: Nat())
213
214  @Axiom
215  def TPred(t1: Term): Unit = {
216    require(t1 :: Nat())
217  } ensuring (Pred(t1) :: Nat())
218
219  //inversion axioms for TPred
220  @Axiom
221  def TPred_inv1(t1: Term): Unit = {
222    require(Pred(t1) :: Nat())
223  } ensuring (t1 :: Nat())
224
225  @Axiom
226  def TPred_inv2(t1: Term, T: Ty): Unit = {
227    require(Pred(t1) :: T)
228  } ensuring (T == Nat())
229
230  @Axiom
231  def Tiszero(t1: Term): Unit = {
232    require(t1 :: Nat())
233  } ensuring (Iszero(t1) :: B())
234
```

```
235  //inversion axioms for Tiszero
236  @Axiom
237  def Tiszero_inv1(t1: Term): Unit = {
238    require(Iszero(t1) :: B())
239  } ensuring (t1 :: Nat())
240
241  @Axiom
242  def Tiszero_inv2(t1: Term, T: Ty): Unit = {
243    require(Iszero(t1) :: T)
244  } ensuring (T == B())
245
246  @Axiom
247  def TPlus(t1: Term, t2: Term): Unit = {
248    require(t1 :: Nat())
249    require(t2 :: Nat())
250  } ensuring (Plus(t1, t2) :: Nat())
251
252  //inversion axioms for TPlus
253  @Axiom
254  def TPlus_inv0(t1: Term, t2: Term, T: Ty): Unit = {
255    require(Plus(t1, t2) :: T)
256  } ensuring (T == Nat())
257
258  @Axiom
259  def TPlus_inv1(t1: Term, t2: Term): Unit = {
260    require(Plus(t1, t2) :: Nat())
261  } ensuring (t1 :: Nat())
262
263  @Axiom
264  def TPlus_inv2(t1: Term, t2: Term): Unit = {
265    require(Plus(t1, t2) :: Nat())
266  } ensuring (t2 :: Nat())
267
268  // steps for soundness proof (progress and preservation) for typed arithmetic expressions
         as given in Pierce, TAPL, Chapter 8
269  @Property
270  def Progress(t: Term, T: Ty): Unit = {
271    require(t :: T)
272    require(!isValue(t))
273  } ensuring (reduce(t) != noTerm())
274
275  @Property
276  def Preservation(t: Term, T: Ty, tres: Term): Unit = {
277    require(t :: T)
278    require(reduce(t) == someTerm(tres))
279  } ensuring (tres :: T)
280
281  @Property
282  def PlusPreservation(t: Term, t1: Term): Unit = {
283    require(t :: Nat())
284    require(t1 :: Nat())
```

```
285   } ensuring (plusop(t, t1) :: Nat())
286
287   @Property
288   def isNVisNat(t: Term): Unit = {
289     require(isNV(t))
290   } ensuring (t :: Nat())
291 }
```

Listing A.1: Full specification of running example of typed arithmetic expressions in ScalaSPL

A.2. A Subset of Typed SQL

```
1  package de.tu_darmstadt.veritas.scalaspl
2
3  import de.tu_darmstadt.veritas.scalaspl.lang.ScalaSPLSpecification
4
5  object SQLSpec extends ScalaSPLSpecification {
6
7    // name of attributes and tables
8    trait Name extends Expression
9
10   // list of attribute names
11   sealed trait AttrL extends Expression
12
13   case class aempty() extends AttrL
14
15   case class acons(hd: Name, tl: AttrL) extends AttrL
16
17   @Recursive(0)
18   def append(atl1: AttrL, atl2: AttrL): AttrL = (atl1, atl2) match {
19     case (aempty(), atl) => atl
20     case (acons(name, atlr), atl) => acons(name, append(atlr, atl))
21   }
22
23   trait FType extends Type
24
25   // type of a table (table schema)
26   sealed trait TType extends Type
27
28   case class ttempty() extends TType
29
30   case class ttcons(n: Name, ft: FType, tt: TType) extends TType
31
32   // Value for a field (underspecified)
33   trait Val extends Expression
34
35   // table row, list of field values (with at least one cell/field per construction!)
36   sealed trait Row extends Expression
37
```

```
38  case class rempty() extends Row
39
40  case class rcons(v: Val, r: Row) extends Row
41
42  // table matrix (list of rows), without "header" (attribute list)
43  sealed trait RawTable extends Expression
44
45  case class tempty() extends RawTable
46
47  case class tcons(r: Row, rt: RawTable) extends RawTable
48
49  // full table with "header" (attribute list)
50  sealed trait Table extends Expression
51
52  case class table(a: AttrL, rt: RawTable) extends Table
53
54  def getRaw(t: Table): RawTable = t match {
55    case table(_, rt) => rt
56  }
57
58  def getAttrL(t: Table): AttrL = t match {
59    case table(al, _) => al
60  }
61
62  // function that assigns a field type to every field value (underspecified)
63  def fieldType(v: Val): FType = ???
64
65  // function that compares whether first field value is smaller than second field value
66  // (underspecified)
67  def lessThan(v1: Val, v2: Val): Boolean = ???
68
69  // function that compares whether first field value is greater than second field value
70  // (underspecified)
71  def greaterThan(v1: Val, v2: Val): Boolean = ???
72
73  // check whether a table corresponds to a given type (functional notation)
74  // does not yet check for whether the table type contains only unique attribute names!!
75  // (but semantics should be possible to define in a sensible way without that
        requirement...)
76  @Static
77  @Recursive(0, 1)
78  def matchingAttrL(tt: TType, attrl: AttrL): Boolean = (tt, attrl) match {
79    case (ttempty(), aempty()) => true
80    case (ttcons(a1, _, ttr), acons(a2, al)) => (a1 == a2) && matchingAttrL(ttr, al)
81    case (_, _) => false
82  }
83
84  @Static
85  @Recursive(0, 1)
86  def welltypedRow(tType: TType, row: Row): Boolean = (tType, row) match {
87    case (ttempty(), rempty()) => true
```

```
88    case (ttcons(_, ft, ttr), rcons(v, r)) => fieldType(v) == ft && welltypedRow(ttr, r)
89    case (_, _) => false
90  }
91
92  @Static
93  @Recursive(1)
94  @Preservable
95  def welltypedRawtable(tty: TType, rt: RawTable): Boolean = (tty, rt) match {
96    case (_, tempty()) => true
97    case (tt, tcons(r, t1)) => welltypedRow(tt, r) && welltypedRawtable(tt, t1)
98  }
99
100 @Static
101 @Preservable
102 def welltypedtable(tty: TType, t: Table): Boolean = (tty, t) match {
103   case (tt, table(al, t1)) => matchingAttrL(tt, al) && welltypedRawtable(tt, t1)
104 }
105
106 //some auxiliary functions on raw tables (all not knowing anything about table types!)
107 //the functions are intended to be used with well−typed tables!!
108
109 @Dynamic
110 @Recursive(1)
111 def rowIn(r: Row, rt: RawTable): Boolean = (r, rt) match {
112   case (_, tempty()) => false
113   case (r1, tcons(r2, rt2)) => (r1 == r2) || rowIn(r1, rt2)
114 }
115
116 //projects a raw table to its first column
117 //returns a raw table with exactly one column or tempty
118 @Dynamic
119 @PreservationProperty("projectFirstRawPreservesWelltypedRaw")
120 @PreservationProperty("projectFirstRawPreservesRowCount")
121 @Recursive(0)
122 def projectFirstRaw(rt: RawTable): RawTable = rt match {
123   case tempty() => tempty()
124   case tcons(rempty(), rt1) => tcons(rempty(), projectFirstRaw(rt1))
125   case tcons(rcons(f, _), rt1) => tcons(rcons(f, rempty()), projectFirstRaw(rt1))
126 }
127
128 //drops the first column of a raw table
129 //returns a raw table with one column less than before or tempty
130 @Dynamic
131 @PreservationProperty("dropFirstColRawPreservesWelltypedRaw")
132 @PreservationProperty("dropFirstColRawPreservesRowCount")
133 @Recursive(0)
134 def dropFirstColRaw(rt: RawTable): RawTable = rt match {
135   case tempty() => tempty()
136   case tcons(rempty(), rt1) => tcons(rempty(), dropFirstColRaw(rt1))
137   case tcons(rcons(_, rr), rt1) => tcons(rr, dropFirstColRaw(rt1))
138 }
```

```
139
140 @FailableType
141 sealed trait OptRawTable
142
143 case class noRawTable() extends OptRawTable
144
145 case class someRawTable(rt: RawTable) extends OptRawTable
146
147 def isSomeRawTable(ort: OptRawTable): Boolean = ort match {
148   case noRawTable() => false
149   case someRawTable(_) => true
150 }
151
152 @Partial
153 def getRawTable(ort: OptRawTable): RawTable = ort match {
154   case someRawTable(rt) => rt
155 }
156
157 @Dynamic
158 @Recursive(0, 1)
159 @Preservable
160 def sameLength(rt1: RawTable, rt2: RawTable): Boolean = (rt1, rt2) match {
161   case (tempty(), tempty()) => true
162   case (tcons(_, tll), tcons(_, tlr)) => sameLength(tll, tlr)
163   case (_, _) => false
164 }
165
166 //attaches a raw table with one column to the front of another raw table
167 //returns a raw table with one column more, possibly not a welltyped one
168 //(if the row counts of the input arguments differ)
169 //assumes that both tables have the same row count!
170 //include empty brackets after tempty such that the parser does not report an error
171 //is treated exactly like tempty for fof-generation
172 @Dynamic
173 @PreservationProperty("attachColToFrontRawPreservesWellTypedRaw")
174 @PreservationProperty("attachColToFrontRawPreservesRowCount")
175 @Recursive(0, 1)
176 def attachColToFrontRaw(rt1: RawTable, rt2: RawTable): RawTable = (rt1, rt2) match {
177   case (tempty(), tempty()) => tempty()
178   case (tcons(rcons(f, rempty()), rt1r), tcons(r, rt2r)) => tcons(rcons(f, r),
        attachColToFrontRaw(rt1r, rt2r))
179   case (_, _) => tcons(rempty(), tempty())
180 }
181
182
183 //definition: union removes duplicate rows
184 //(but only between the two tables, not within a table!)
185 //preserves row order of the two original raw tables
186 @Dynamic
187 @PreservationProperty("rawUnionPreservesWellTypedRaw")
188 @Recursive(0)
```

```
189  def rawUnion(rtab1: RawTable, rtab2: RawTable): RawTable = (rtab1, rtab2) match {
190    case (tempty(), rt) => rt
191    case (tcons(r, rtr), rt1) =>
192      val urt1rt2 = rawUnion(rtr, rt1)
193      if (!rowIn(r, rt1))
194        tcons(r, urt1rt2)
195      else
196        urt1rt2
197  }
198
199  @Dynamic
200  @PreservationProperty("rawIntersectionPreservesWellTypedRaw")
201  @Recursive(0)
202  def rawIntersection(rtab1: RawTable, rtab2: RawTable): RawTable = (rtab1, rtab2)
       match {
203    case (tempty(), _) => tempty()
204    case (tcons(r, tempty()), rt1) =>
205      if (rowIn(r, rt1))
206        tcons(r, tempty())
207      else
208        tempty()
209    case (tcons(r, rtr), rt1) =>
210      val irt1rt2 = rawIntersection(rtr, rt1)
211      if (rowIn(r, rt1))
212        tcons(r, irt1rt2)
213      else irt1rt2
214  }
215
216  @Dynamic
217  @PreservationProperty("rawDifferencePreservesWellTypedRaw")
218  @Recursive(0)
219  def rawDifference(rtab1: RawTable, rtab2: RawTable): RawTable = (rtab1, rtab2) match {
220    case (tempty(), _) => tempty()
221    case (tcons(r, tempty()), rt2) =>
222      if (!rowIn(r, rt2))
223        tcons(r, tempty())
224      else tempty()
225    case (tcons(r, rtr), rt2) =>
226      val drt1rt2 = rawDifference(rtr, rt2)
227      if (!rowIn(r, rt2))
228        tcons(r, drt1rt2)
229      else drt1rt2
230  }
231
232  @FailableType
233  sealed trait OptTable
234
235  case class noTable() extends OptTable
236
237  case class someTable(t: Table) extends OptTable
238
```

```
239 def isSomeTable(ot: OptTable): Boolean = ot match {
240   case noTable() => false
241   case someTable(_) => true
242 }
243
244 @Partial
245 def getTable(ot: OptTable): Table = ot match {
246   case someTable(t) => t
247 }
248
249
250 sealed trait TStore
251
252 case class emptyStore() extends TStore
253
254 case class bindStore(n: Name, t: Table, rst: TStore) extends TStore
255
256 @Dynamic
257 @ProgressProperty("successfulLookup")
258 @PreservationProperty("welltypedLookup") // FIXME: In the strict sense,
      ''welltypedLookup'' is no preservation lemma
259 @Recursive(1)
260 def lookupStore(an: Name, tst: TStore): OptTable = (an, tst) match {
261   case (_, emptyStore()) => noTable()
262   case (n, bindStore(m, t, tsr)) =>
263     if (n == m)
264       someTable(t)
265     else lookupStore(n, tsr)
266 }
267
268 sealed trait TTContext extends Context
269
270 case class emptyContext() extends TTContext
271
272 case class bindContext(n: Name, tt: TType, ttr: TTContext) extends TTContext
273
274
275 @FailableType
276 sealed trait OptTType
277
278 case class noTType() extends OptTType
279
280 case class someTType(tt: TType) extends OptTType
281
282 def isSomeTType(ott: OptTType): Boolean = ott match {
283   case noTType() => false
284   case someTType(_) => true
285 }
286
287 @Partial
288 def getTType(ott: OptTType): TType = ott match {
```

```
289     case someTType(tt) => tt
290   }
291
292   @Static
293   @Recursive(1)
294   def lookupContext(an: Name, ttc: TTContext): OptTType = (an, ttc) match {
295     case (_, emptyContext()) => noTType()
296     case (n, bindContext(m, tt, ttcr)) =>
297       if (n == m)
298         someTType(tt)
299       else lookupContext(n, ttcr)
300   }
301
302   sealed trait Exp extends Expression
303
304   case class constant(v: Val) extends Exp
305
306   case class lookup(n: Name) extends Exp
307
308   //predicates for where clauses of queries
309   sealed trait Pred extends Expression
310
311   case class ptrue() extends Pred
312
313   case class and(p1: Pred, p2: Pred) extends Pred
314
315   case class not(p: Pred) extends Pred
316
317   case class eq(e1: Exp, e2: Exp) extends Pred
318
319   case class gt(e1: Exp, e2: Exp) extends Pred
320
321   case class lt(e1: Exp, e2: Exp) extends Pred
322
323   // Query syntax
324   sealed trait Select extends Expression
325
326   case class all() extends Select
327
328   case class list(attrL: AttrL) extends Select
329
330
331   sealed trait Query extends Expression
332
333   case class tvalue(t: Table) extends Query
334
335   case class selectFromWhere(s: Select, name: Name, pred: Pred) extends Query
336
337   case class Union(q1: Query, q2: Query) extends Query
338
339   case class Intersection(q1: Query, q2: Query) extends Query
```

```
340
341  case class Difference(q1: Query, q2: Query) extends Query
342
343  def isValue(q: Query): Boolean = q match {
344    case tvalue(_) => true
345    case selectFromWhere(_, _, _) => false
346    case Union(_, _) => false
347    case Intersection(_, _) => false
348    case Difference(_, _) => false
349  }
350
351
352  //functions for semantics of SQL
353  @FailableType
354  sealed trait OptQuery
355
356  case class noQuery() extends OptQuery
357
358  case class someQuery(q: Query) extends OptQuery
359
360  def isSomeQuery(oq: OptQuery): Boolean = oq match {
361    case noQuery() => false
362    case someQuery(_) => true
363  }
364
365  @Partial
366  def getQuery(oq: OptQuery): Query = oq match {
367    case someQuery(q) => q
368  }
369
370  @Dynamic
371  @ProgressProperty("findColTypeImpliesfindCol")
372  @PreservationProperty("findColPreservesWelltypedRaw")
373  @PreservationProperty("findColPreservesRowCount")
374  @Recursive(1)
375  def findCol(a: Name, attrL: AttrL, rt: RawTable): OptRawTable = (a, attrL, rt) match {
376    case (n, aempty(), _) => noRawTable()
377    case (n, acons(n1, alr), rtr) =>
378      if (n == n1)
379        someRawTable(projectFirstRaw(rtr))
380      else
381        findCol(n, alr, dropFirstColRaw(rtr))
382  }
383
384  // for projection base case: projecting on an empty attribute list must yield a
385  // table with as many empty rows as the rowcount of the given table
386  @Dynamic
387  @PreservationProperty("welltypedEmptyProjection")
388  @PreservationProperty("projectEmptyColPreservesRowCount")
389  @Recursive(0)
390  def projectEmptyCol(rt: RawTable): RawTable = rt match {
```

```
391     case tempty() => tempty()
392     case tcons(_, t) => tcons(rempty(), projectEmptyCol(t))
393   }
394
395   // arguments: select-list table-list table-rows
396   @Dynamic
397   @ProgressProperty("projectColsProgress")
398   @PreservationProperty("projectColsWelltypedWithSelectType")
399   @PreservationProperty("projectColsPreservesRowCount")
400   @Recursive(0)
401   def projectCols(attrl1: AttrL, attrl2: AttrL, rtable: RawTable): OptRawTable = (attrl1,
          attrl2, rtable) match {
402     case (aempty(), _, rt) => someRawTable(projectEmptyCol(rt))
403     case (acons(n, al2), al1, rt) =>
404       val col = findCol(n, al1, rt)
405       val rest = projectCols(al2, al1, rt)
406       if (isSomeRawTable(col) && isSomeRawTable(rest))
407         someRawTable(attachColToFrontRaw(getRawTable(col), getRawTable(rest)))
408       else
409         noRawTable()
410   }
411
412   @Dynamic
413   @ProgressProperty("projectTableProgress")
414   @PreservationProperty("projectTableWelltypedWithSelectType")
415   //@PreservationProperty("projectTypeAttrLMatchesAttrL") // FIXME:
          projectTypeAttrLMatchesAttrL is no preservation lemma
416   def projectTable(s: Select, tab: Table): OptTable = (s, tab) match {
417     case (all(), t) => someTable(t)
418     case (list(al), t) =>
419       val projected = projectCols(al, getAttrL(t), getRaw(t))
420       if (isSomeRawTable(projected))
421         someTable(table(al, getRawTable(projected)))
422       else
423         noTable()
424   }
425
426   @FailableType
427   sealed trait OptVal
428
429   case class noVal() extends OptVal
430
431   case class someVal(v: Val) extends OptVal
432
433   def isSomeVal(ov: OptVal): Boolean = ov match {
434     case noVal() => false
435     case someVal(_) => true
436   }
437
438   @Partial
439   def getVal(ov: OptVal): Val = ov match {
```

```
440     case someVal(v) => v
441   }
442
443   @Dynamic
444   def evalExpRow(e: Exp, attrL: AttrL, row: Row): OptVal = (e, attrL, row) match {
445     case (constant(v), _, _) => someVal(v)
446     case (lookup(a), acons(a2, al), rcons(v, r)) =>
447       if (a == a2)
448         someVal(v)
449       else
450         evalExpRow(lookup(a), al, r)
451     case (_, _, _) => noVal()
452   }
453
454   // returns true iff predicate succeeds on row
455   // returns false if predicate evaluates to false or if predicate evaluation fails
456   @Dynamic
457   @Recursive(0)
458   def filterSingleRow(p: Pred, attrL: AttrL, row: Row): Boolean = (p, attrL, row) match {
459     case (ptrue(), _, _) => true
460     case (and(p1, p2), al, r) => filterSingleRow(p1, al, r) && filterSingleRow(p2, al, r)
461     case (not(pr), al, r) => !filterSingleRow(pr, al, r)
462     case (eq(e1, e2), al, r) =>
463       val v1 = evalExpRow(e1, al, r)
464       val v2 = evalExpRow(e2, al, r)
465       isSomeVal(v1) && isSomeVal(v2) && getVal(v1) == getVal(v2)
466     case (gt(e1, e2), al, r) =>
467       val v1 = evalExpRow(e1, al, r)
468       val v2 = evalExpRow(e2, al, r)
469       isSomeVal(v1) && isSomeVal(v2) && greaterThan(getVal(v1), getVal(v2))
470     case (lt(e1, e2), al, r) =>
471       val v1 = evalExpRow(e1, al, r)
472       val v2 = evalExpRow(e2, al, r)
473       isSomeVal(v1) && isSomeVal(v2) && lessThan(getVal(v1), getVal(v2))
474   }
475
476   // filter rows that satisfy pred
477   @Dynamic
478   @PreservationProperty("filterRowsPreservesTable")
479   @Recursive(0)
480   def filterRows(rt: RawTable, attrL: AttrL, pred: Pred): RawTable = (rt, attrL, pred)
        match {
481     case (tempty(), _, _) => tempty()
482     case (tcons(r, rtr), al, p) =>
483       val rts = filterRows(rtr, al, p)
484       if (filterSingleRow(p, al, r))
485         tcons(r, rts)
486       else
487         rts
488   }
489
```

```
490  @Dynamic
491  @PreservationProperty("filterPreservesType")
492  def filterTable(t: Table, pred: Pred): Table = (t, pred) match {
493    case (table(al, rt), p) => table(al, filterRows(rt, al, p))
494  }
495
496  @Dynamic
497  @ProgressProperty("Progress")
498  @PreservationProperty("Preservation")
499  @Recursive(0)
500  def reduce(query: Query, tst: TStore): OptQuery = (query, tst) match {
501    case (tvalue(_), _) => noQuery()
502    case (selectFromWhere(s, n, p), ts) =>
503      val maybeTable = lookupStore(n, ts)
504      if (isSomeTable(maybeTable)) {
505        val filtered = filterTable(getTable(maybeTable), p)
506        val maybeSelected = projectTable(s, filtered)
507        if (isSomeTable(maybeSelected))
508          someQuery(tvalue(getTable(maybeSelected)))
509        else noQuery()
510      }
511      else
512        noQuery()
513    case (Union(tvalue(t1), tvalue(t2)), ts) =>
514      someQuery(tvalue(table(getAttrL(t1), rawUnion(getRaw(t1), getRaw(t2)))))
515    case (Union(tvalue(t), q2), ts) =>
516      val q2reduce = reduce(q2, ts)
517      if (isSomeQuery(q2reduce))
518        someQuery(Union(tvalue(t), getQuery(q2reduce)))
519      else
520        noQuery()
521    case (Union(q1, q2), ts) =>
522      val q1reduce = reduce(q1, ts)
523      if (isSomeQuery(q1reduce))
524        someQuery(Union(getQuery(q1reduce), q2))
525      else
526        noQuery()
527    case (Intersection(tvalue(t1), tvalue(t2)), ts) =>
528      someQuery(tvalue(table(getAttrL(t1), rawIntersection(getRaw(t1), getRaw(t2)))))
529    case (Intersection(tvalue(t), q2), ts) =>
530      val q2reduce = reduce(q2, ts)
531      if (isSomeQuery(q2reduce))
532        someQuery(Intersection(tvalue(t), getQuery(q2reduce)))
533      else
534        noQuery()
535    case (Intersection(q1, q2), ts) =>
536      val q1reduce = reduce(q1, ts)
537      if (isSomeQuery(q1reduce))
538        someQuery(Intersection(getQuery(q1reduce), q2))
539      else
540        noQuery()
```

```
541     case (Difference(tvalue(t1), tvalue(t2)), ts) =>
542       someQuery(tvalue(table(getAttrL(t1), rawDifference(getRaw(t1), getRaw(t2)))))
543     case (Difference(tvalue(t), q2), ts) =>
544       val q2reduce = reduce(q2, ts)
545       if (isSomeQuery(q2reduce))
546         someQuery(Difference(tvalue(t), getQuery(q2reduce)))
547       else
548         noQuery()
549     case (Difference(q1, q2), ts) =>
550       val q1reduce = reduce(q1, ts)
551       if (isSomeQuery(q1reduce))
552         someQuery(Difference(getQuery(q1reduce), q2))
553       else
554         noQuery()
555   }
556
557   @FailableType
558   sealed trait OptFType
559
560   case class noFType() extends OptFType
561
562   case class someFType(ft: FType) extends OptFType
563
564   def isSomeFType(oft: OptFType): Boolean = oft match {
565     case noFType() => false
566     case someFType(a) => true
567   }
568
569   @Partial
570   def getFType(oft: OptFType): FType = oft match {
571     case someFType(a) => a
572   }
573
574   @Static
575   @Recursive(1)
576   def findColType(an: Name, tt: TType): OptFType = (an, tt) match {
577     case (n, ttempty()) => noFType()
578     case (n, ttcons(a, ft, ttr)) =>
579       if (n == a)
580         someFType(ft)
581       else
582         findColType(n, ttr)
583   }
584
585   @Static
586   @Recursive(0)
587   @AuxiliaryProperty("projectTypeAttrLMatchesAttrL")
588   def projectTypeAttrL(attrl: AttrL, tty: TType): OptTType = (attrl, tty) match {
589     case (aempty(), tt) => someTType(ttempty())
590     case (acons(a, alr), tt) =>
591       val ft = findColType(a, tt)
```

```
592       val tprest = projectTypeAttrL(alr, tt)
593       if (isSomeFType(ft) && isSomeTType(tprest))
594         someTType(ttcons(a, getFType(ft), getTType(tprest)))
595       else
596         noTType()
597   }
598
599   @Static
600   def projectType(sel: Select, tt: TType): OptTType = (sel, tt) match {
601     case (all(), tt1) => someTType(tt1)
602     case (list(al), tt1) => projectTypeAttrL(al, tt1)
603   }
604
605   @Static
606   @Recursive(0)
607   def typeOfExp(e: Exp, tty: TType): OptFType = (e, tty) match {
608     case (constant(fv), tt) => someFType(fieldType(fv))
609     case (lookup(n), ttempty()) => noFType()
610     case (lookup(n), ttcons(a2, ft, ttr)) =>
611       if (n == a2)
612         someFType(ft)
613       else
614         typeOfExp(lookup(n), ttr)
615   }
616
617
618   @Static
619   @Recursive(0)
620   def tcheckPred(pred: Pred, tType: TType): Boolean = (pred, tType) match {
621     case (ptrue(), tt) => true
622     case (and(p1, p2), tt) => tcheckPred(p1, tt) && tcheckPred(p2, tt)
623     case (not(p), tt) => tcheckPred(p, tt)
624     case (eq(e1, e2), tt) =>
625       val t1 = typeOfExp(e1, tt)
626       val t2 = typeOfExp(e2, tt)
627       isSomeFType(t1) && isSomeFType(t2) && (getFType(t1) == getFType(t2))
628     case (gt(e1, e2), tt) =>
629       val t1 = typeOfExp(e1, tt)
630       val t2 = typeOfExp(e2, tt)
631       isSomeFType(t1) && isSomeFType(t2) && (getFType(t1) == getFType(t2))
632     case (lt(e1, e2), tt) =>
633       val t1 = typeOfExp(e1, tt)
634       val t2 = typeOfExp(e2, tt)
635       isSomeFType(t1) && isSomeFType(t2) && (getFType(t1) == getFType(t2))
636   }
637
638   //axioms on behavior of table type context
639   @Axiom
640   def TTTContextDuplicate(x: Name, y: Name, Tx: TType, Ty: TType, C: TTContext, e:
        Query, T: TType): Unit = {
641     require(x == y)
```

```
642     require(bindContext(x, Tx, bindContext(y, Ty, C)) |− e :: T)
643   } ensuring (bindContext(x, Tx, C) |− e :: T)
644
645   @Axiom
646   def TTTContextSwap(x: Name, y: Name, Tx: TType, Ty: TType, C: TTContext, e:
          Query, T: TType): Unit = {
647     require(x != y)
648     require(bindContext(x, Tx, bindContext(y, Ty, C)) |− e :: T)
649   } ensuring(bindContext(y, Ty, bindContext(x, Tx, C)) |− e :: T)
650
651   @Axiom
652   def Ttvalue(t: Table, TTC: TTContext, TT: TType): Unit = {
653     require(welltypedtable(TT, t))
654   } ensuring(TTC |− tvalue(t) :: TT)
655
656   @Axiom
657   def TSelectFromWhere(tn: Name, TTC: TTContext, TT: TType, p: Pred, sel: Select,
          TTr: TType): Unit = {
658     require(lookupContext(tn, TTC) == someTType(TT))
659     require(tcheckPred(p, TT))
660     require(projectType(sel, TT) == someTType(TTr))
661   } ensuring(TTC |− selectFromWhere(sel, tn, p) :: TTr)
662
663   @Axiom
664   def TUnion(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
665     require(TTC |− q1 :: TT)
666     require(TTC |− q2 :: TT)
667   } ensuring(TTC |− Union(q1, q2) :: TT)
668
669   @Axiom
670   def TIntersection(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
671     require(TTC |− q1 :: TT)
672     require(TTC |− q2 :: TT)
673   } ensuring(TTC |− Intersection(q1, q2) :: TT)
674
675   @Axiom
676   def TDifference(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
677     require(TTC |− q1 :: TT)
678     require(TTC |− q2 :: TT)
679   } ensuring(TTC |− Difference(q1, q2) :: TT)
680
681   // type inversion axioms
682   @Axiom
683   def Ttvalue_inv(t: table, TTC: TTContext, TT: TType): Unit = {
684     require(TTC |− tvalue(t) :: TT)
685   } ensuring(welltypedtable(TT, t))
686
687   @Axiom
688   def TSelectFromWhere_inv(tn: Name, TTC: TTContext, p: Pred, sel: Select, TTr:
          TType): Unit = {
689     require(TTC |− selectFromWhere(sel, tn, p) :: TTr)
```

```
690   } ensuring(exists((TT: TType) => lookupContext(tn, TTC) == someTType(TT) &&
          tcheckPred(p, TT) && projectType(sel, TT) == someTType(TTr)))
691
692   @Axiom
693   def TUnion_inv1(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
694     require(TTC |− Union(q1, q2) :: TT)
695   } ensuring(TTC |− q1 :: TT)
696
697   @Axiom
698   def TUnion_inv2(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
699     require(TTC |− Union(q1, q2) :: TT)
700   } ensuring(TTC |− q2 :: TT)
701
702   @Axiom
703   def TIntersection_inv1(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
704     require(TTC |− Intersection(q1, q2) :: TT)
705   } ensuring(TTC |− q1 :: TT)
706
707   @Axiom
708   def TIntersection_inv2(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
709     require(TTC |− Intersection(q1, q2) :: TT)
710   } ensuring(TTC |− q2 :: TT)
711
712   @Axiom
713   def TDifference_inv1(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
714     require(TTC |− Difference(q1, q2) :: TT)
715   } ensuring(TTC |− q1 :: TT)
716
717   @Axiom
718   def TDifference_inv2(q1: Query, q2: Query, TT: TType, TTC: TTContext): Unit = {
719     require(TTC |− Difference(q1, q2) :: TT)
720   } ensuring(TTC |− q2 :: TT)
721
722
723   // determines whether a given TTContext is consistent with a given TStore
724   // and whether the table in the store is well−typed with regard to the table type in the
          context
725   // design decision: require bindings to appear in exactly the SAME ORDER! (simpler?)
726   @Static
727   @Recursive(0, 1)
728   def storeContextConsistent(ts: TStore, ttc: TTContext): Boolean = (ts, ttc) match {
729     case (emptyStore(), emptyContext()) => true
730     case (bindStore(tn1, t, tsr), bindContext(tn2, tt, ttcr)) =>
731       tn1 == tn2 && welltypedtable(tt, t) && storeContextConsistent(tsr, ttcr)
732     case (_, _) => false
733   }
734
735   // LEMMAS BEGIN
736   //PROGRESS
737   @Property
738   def Progress(ts: TStore, ttc: TTContext, q: Query): Unit = {
```

```
739     require(storeContextConsistent(ts, ttc))
740     require(!isValue(q))
741     require(exists((tt1: TType) => ttc |- q :: tt1))
742   } ensuring exists((qr: Query) => reduce(q, ts) == someQuery(qr))
743
744   // auxiliary lemmas for progress proof
745   @Property
746   def successfulLookup(ttc: TTContext, ts: TStore, ref: Name, tt: TType): Unit = {
747     require(storeContextConsistent(ts, ttc))
748     require(lookupContext(ref, ttc) == someTType(tt))
749   } ensuring exists((t: Table) => lookupStore(ref, ts) == someTable(t))
750
751   @Property
752   def welltypedLookup(ttc: TTContext, ts: TStore, ref: Name, tt: TType, t: Table): Unit = {
753     require(storeContextConsistent(ts, ttc))
754     require(lookupContext(ref, ttc) == someTType(tt))
755     require(lookupStore(ref, ts) == someTable(t))
756   } ensuring welltypedtable(tt, t)
757
758   @Property
759   def filterPreservesType(tt: TType, t: Table, result: Table, p: Pred): Unit = {
760     require(welltypedtable(tt, t))
761     require(filterTable(t, p) == result)
762   } ensuring welltypedtable(tt, result)
763
764   @Property
765   def projectTableProgress(tt: TType, t: Table, s: Select, tt2: TType): Unit = {
766     require(welltypedtable(tt, t))
767     require(projectType(s, tt) == someTType(tt2))
768   } ensuring exists((t2: Table) => projectTable(s, t) == someTable(t2))
769
770   @Property
771   def filterRowsPreservesTable(tt: TType, rt: RawTable, rt2: RawTable, al: AttrL, p: Pred):
          Unit = {
772     require(welltypedRawtable(tt, rt))
773     require(filterRows(rt, al, p) == rt2)
774   } ensuring welltypedRawtable(tt, rt2)
775
776   @Property
777   def projectColsProgress(tt: TType, alt: AttrL, rt: RawTable, al: AttrL, tt2: TType): Unit =
          {
778     require(welltypedRawtable(tt, rt))
779     require(matchingAttrL(tt, alt))
780     //require(projectType(list(al2), tt) == someTType(tt2)) NOTE: expanded this to:
781     require(projectTypeAttrL(al, tt) == someTType(tt2))
782   } ensuring exists((rt2: RawTable) => projectCols(al, alt, rt) == someRawTable(rt2))
783
784   @Property
785   def findColTypeImpliesfindCol(tt: TType, al: AttrL, rt: RawTable, n: Name, ft: FType):
          Unit = {
786     require(welltypedRawtable(tt, rt))
```

```
787     require(matchingAttrL(tt, al))
788     require(findColType(n, tt) == someFType(ft))
789   } ensuring exists((rt2: RawTable) => findCol(n, al, rt) == someRawTable(rt2))
790
791   @Property
792   def dropFirstColRawPreservesWelltypedRaw(tt: TType, n: Name, ft: FType, ttrest: TType,
          rt: RawTable, rt1: RawTable): Unit = {
793     require(tt == ttcons(n, ft, ttrest)) // |tt| > 0
794     require(welltypedRawtable(tt, rt))
795     require(dropFirstColRaw(rt) == rt1)
796   } ensuring welltypedRawtable(ttrest, rt1)
797
798   //PRESERVATION
799
800   // union, intersection, difference preserve well-typedness of raw tables
801   @Property
802   def rawUnionPreservesWellTypedRaw(rt: RawTable, rt1: RawTable, result: RawTable, tt:
          TType): Unit = {
803     require(welltypedRawtable(tt, rt))
804     require(welltypedRawtable(tt, rt1))
805     require(rawUnion(rt, rt1) == result)
806   } ensuring welltypedRawtable(tt, result)
807
808   @Property
809   def rawIntersectionPreservesWellTypedRaw(rt: RawTable, rt1: RawTable, result:
          RawTable, tt: TType): Unit = {
810     require(welltypedRawtable(tt, rt))
811     require(welltypedRawtable(tt, rt1))
812     require(rawIntersection(rt, rt1) == result)
813   } ensuring welltypedRawtable(tt, result)
814
815   @Property
816   def rawDifferencePreservesWellTypedRaw(rt: RawTable, rt1: RawTable, result: RawTable,
          tt: TType): Unit = {
817     require(welltypedRawtable(tt, rt))
818     require(welltypedRawtable(tt, rt1))
819     require(rawDifference(rt, rt1) == result)
820   } ensuring welltypedRawtable(tt, result)
821
822   @Property
823   def projectTypeAttrLMatchesAttrL(al: AttrL, tt: TType, tt2: TType): Unit = {
824     require(projectTypeAttrL(al, tt) == someTType(tt2))
825   } ensuring matchingAttrL(tt2, al)
826
827   @Property
828   def welltypedEmptyProjection(rt: RawTable, rt1: RawTable, tt: TType): Unit = {
829     require(tt == ttempty())
830     require(projectEmptyCol(rt) == rt1)
831   } ensuring welltypedRawtable(tt, rt1)
832
833   @Property
```

```
834  def projectFirstRawPreservesWelltypedRaw(rt: RawTable, rt1: RawTable,
835                                           tt: TType, tt1: TType,
836                                           a: Name, ct: FType, ttrest: TType): Unit = {
837    require(tt == ttcons(a, ct, ttrest))
838    require(tt1 == ttcons(a, ct, ttempty()))
839    require(welltypedRawtable(tt, rt))
840    require(projectFirstRaw(rt) == rt1)
841  } ensuring welltypedRawtable(tt1, rt1)
842
843  @Property
844  def findColPreservesWelltypedRaw(n: Name, al: AttrL, rt: RawTable,
845                                   tt: TType, tt2: TType, ft: FType, rt2: RawTable): Unit
                                       = {
846    require(welltypedRawtable(tt, rt))
847    require(matchingAttrL(tt, al))
848    require(findColType(n, tt) == someFType(ft))
849    require(findCol(n, al, rt) == someRawTable(rt2))
850    require(tt2 == ttcons(n, ft, ttempty()))
851  } ensuring welltypedRawtable(tt2, rt2)
852
853  @Property
854  def attachColToFrontRawPreservesWellTypedRaw(tt1: TType, name1: Name, ft1: FType,
855                                               tt2: TType, tt3: TType,
856                                               rt: RawTable, rt1: RawTable, rt2:
                                                     RawTable): Unit = {
857    // |tt1| == 1
858    require(tt1 == ttcons(name1, ft1, ttempty()))
859    require(sameLength(rt, rt1))
860    require(welltypedRawtable(tt1, rt))
861    require(welltypedRawtable(tt2, rt1))
862    require(attachColToFrontRaw(rt, rt1) == rt2)
863    require(tt3 == ttcons(name1, ft1, tt2))
864  } ensuring welltypedRawtable(tt3, rt2)
865
866  @Property
867  def attachColToFrontRawPreservesRowCount(tt1: TType, a: Name, ct: FType,
868                                           rt: RawTable, rt1: RawTable, rt2: RawTable):
                                               Unit = {
869    require(tt1 == ttcons(a, ct, ttempty()))
870    require(welltypedRawtable(tt1, rt))
871    require(sameLength(rt, rt1))
872    require(attachColToFrontRaw(rt, rt1) == rt2)
873  } ensuring sameLength(rt, rt2)
874
875  @Property
876  def projectFirstRawPreservesRowCount(rt: RawTable, rt1: RawTable): Unit = {
877    require(projectFirstRaw(rt) == rt1)
878  } ensuring sameLength(rt, rt1)
879
880  @Property
881  def dropFirstColRawPreservesRowCount(rt: RawTable, rt1: RawTable): Unit = {
```

```
882     require(dropFirstColRaw(rt) == rt1)
883   } ensuring sameLength(rt, rt1)
884
885   @Property
886   def findColPreservesRowCount(n: Name, al: AttrL, rt: RawTable, rt1: RawTable): Unit = {
887     require(findCol(n, al, rt) == someRawTable(rt1))
888   } ensuring sameLength(rt, rt1)
889
890   @Property
891   def projectEmptyColPreservesRowCount(rt: RawTable, rt1: RawTable): Unit = {
892     require(projectEmptyCol(rt) == rt1)
893   } ensuring sameLength(rt, rt1)
894
895   @Property
896   def projectColsPreservesRowCount(tt: TType, tt1: TType, al: AttrL, al1: AttrL, rt:
          RawTable, rt1: RawTable): Unit = {
897     require(projectTypeAttrL(al, tt) == someTType(tt1))
898     require(projectCols(al, al1, rt) == someRawTable(rt1))
899     require(welltypedRawtable(tt, rt))
900     require(matchingAttrL(tt, al1))
901   } ensuring sameLength(rt, rt1)
902
903   @Property
904   def projectColsWelltypedWithSelectType(al: AttrL, tal: AttrL, rt: RawTable, rt1:
          RawTable, tt: TType, tt1: TType): Unit = {
905     require(welltypedRawtable(tt, rt))
906     require(matchingAttrL(tt, tal))
907     require(projectTypeAttrL(al, tt) == someTType(tt1))
908     require(projectCols(al, tal, rt) == someRawTable(rt1))
909   } ensuring welltypedRawtable(tt1, rt1)
910
911   @Property
912   def projectTableWelltypedWithSelectType(s: Select, t: Table, t1: Table, tt: TType, tt1:
          TType): Unit = {
913     require(welltypedtable(tt, t))
914     require(projectType(s, tt) == someTType(tt1))
915     require(projectTable(s, t) == someTable(t1))
916   } ensuring welltypedtable(tt1, t1)
917
918   @Property
919   def Preservation(ttc: TTContext, ts: TStore, q: Query, qr: Query, tt: TType): Unit = {
920     require(storeContextConsistent(ts, ttc))
921     require(ttc |- q :: tt)
922     require(reduce(q, ts) == someQuery(qr))
923   } ensuring(ttc |- qr :: tt)
924   // LEMMAS END
925 }
```

Listing A.2: Full specification of typed SQL case study in ScalaSPL

A.3. A Typed Questionnaire Language (QL)

```
1  package de.tu_darmstadt.veritas.scalaspl
2
3  import de.tu_darmstadt.veritas.scalaspl.lang.ScalaSPLSpecification
4
5  object QLSpec extends ScalaSPLSpecification {
6
7    // BasicTypes
8    sealed trait YN extends Expression
9    case class yes() extends YN
10   case class no() extends YN
11
12   def and(a: YN, b: YN): YN = (a, b) match {
13     case (yes(), yes()) => yes()
14     case (_, _) => no()
15   }
16
17   def or(a: YN, b: YN): YN = (a, b) match {
18     case (no(), no()) => no()
19     case (_, _) => yes()
20   }
21
22   def not(a: YN): YN = a match {
23     case yes() => no()
24     case no() => yes()
25   }
26
27   sealed trait nat extends Expression
28   case class zero() extends nat
29   case class succ(n: nat) extends nat
30
31   def pred(n: nat): nat = n match {
32     case zero() => zero()
33     case succ(n) => n
34   }
35
36   def gt(a: nat, b: nat): YN = (a, b) match {
37     case (zero(), _) => no()
38     case (succ(_), zero()) => yes()
39     case (succ(n1), succ(n2)) => gt(n1, n1)
40   }
41
42   def lt(a: nat, b: nat): YN = (a, b) match {
43     case (_, zero()) => no()
44     case (zero(), succ(_)) => yes()
45     case (succ(n1), succ(n2)) => lt(n1, n1)
46   }
47
48   def plus(a: nat, b: nat): nat = (a, b) match {
49     case (n, zero()) => n
```

```
50    case (n1, succ(n2)) => succ(plus(n1, n2))
51  }
52
53  def minus(a: nat, b: nat): nat = (a, b) match {
54    case (n, zero()) => n
55    case (n1, succ(n2)) => pred(minus(n1, n2))
56  }
57
58  def multiply(a: nat, b: nat): nat = (a, b) match {
59    case (_, zero()) => zero()
60    case (n1, succ(n2)) => plus(n1, multiply(n1, n2))
61  }
62
63  def divide(a: nat, b: nat): nat = (a, b) match {
64    case (n1, n2) =>
65      if(gt(n1, n2) == yes())
66        succ(divide(minus(n1, n2), n2))
67      else
68        zero()
69  }
70
71  trait char extends Expression
72
73  sealed trait string extends Expression
74  case class sempty() extends string
75  case class scons(c: char, tail: string) extends string
76
77  // QLSyntax
78  trait QID extends Expression
79
80  trait GID extends Expression
81
82  trait Label extends Expression
83
84  sealed trait Aval extends Expression
85  case class B(value: YN) extends Aval
86  case class Num(value: nat) extends Aval
87  case class T(value: string) extends Aval
88
89  @FailableType
90  sealed trait OptAval
91  case class noAval() extends OptAval
92  case class someAval(value: Aval) extends OptAval
93
94  def isSomeAval(opt: OptAval): Boolean = opt match {
95    case noAval() => false
96    case someAval(_) => true
97  }
98
99  @Partial
100 def getAval(opt: OptAval): Aval = opt match {
```

```
101     case someAval(aval) => aval
102   }
103
104   sealed trait AType extends Expression with Type
105   case class YesNo() extends AType
106   case class Number() extends AType
107   case class Text() extends AType
108
109   @FailableType
110   sealed trait OptAType
111   case class noAType() extends OptAType
112   case class someAType(typ: AType) extends OptAType
113
114   def isSomeAType(opt: OptAType): Boolean = opt match {
115     case noAType() => false
116     case someAType(_) => true
117   }
118
119   @Partial
120   def getAType(opt: OptAType): AType = opt match {
121     case someAType(atype) => atype
122   }
123
124   @Static
125   def typeOf(aval: Aval): AType = aval match {
126     case B(_) => YesNo()
127     case Num(_) => Number()
128     case T(_) => Text()
129   }
130
131   sealed trait ATList extends Expression
132   case class atempty() extends ATList
133   case class atcons(atype: AType, rem: ATList) extends ATList
134
135   def append(atl1: ATList, atl2: ATList): ATList = (atl1, atl2) match {
136     case (atempty(), atl) => atl
137     case (atcons(atype, atlr), atl) => atcons(atype, append(atlr, atl))
138   }
139
140   sealed trait BinOpT extends Expression
141   case class addop() extends BinOpT
142   case class subop() extends BinOpT
143   case class mulop() extends BinOpT
144   case class divop() extends BinOpT
145   case class eqop() extends BinOpT
146   case class gtop() extends BinOpT
147   case class ltop() extends BinOpT
148   case class andop() extends BinOpT
149   case class orop() extends BinOpT
150
151   sealed trait UnOpT extends Expression
```

```
152 case class notop() extends UnOpT
153
154 sealed trait Exp extends Expression
155 case class constant(aval: Aval) extends Exp
156 case class qvar(qid: QID) extends Exp
157 case class binop(e1: Exp, op: BinOpT, e2: Exp) extends Exp
158 case class unop(op: UnOpT, e: Exp) extends Exp
159
160 sealed trait Entry extends Expression
161 case class question(qid: QID, l: Label, at: AType) extends Entry
162 case class value(qid: QID, at: AType, exp: Exp) extends Entry
163 case class defquestion(qid: QID, l: Label, at: AType) extends Entry
164 case class ask(qid: QID) extends Entry
165
166 sealed trait Questionnaire extends Expression
167 case class qempty() extends Questionnaire
168 case class qsingle(entry: Entry) extends Questionnaire
169 case class qseq(qs1: Questionnaire, qs2: Questionnaire) extends Questionnaire
170 case class qcond(e: Exp, thn: Questionnaire, els: Questionnaire) extends Questionnaire
171 case class qgroup(gid: GID, qs: Questionnaire) extends Questionnaire
172
173 // QLSemanticsData
174
175 sealed trait AnsMap extends Expression
176 case class aempty() extends AnsMap
177 case class abind(qid: QID, aval: Aval, al: AnsMap) extends AnsMap
178
179 @Dynamic
180 @ProgressProperty("lookupAnsMapProgress")
181 @PreservationProperty("lookupAnsMapPreservation")
182 @Recursive(1)
183 def lookupAnsMap(id: QID, am: AnsMap): OptAval = (id, am) match {
184   case (_, aempty()) => noAval()
185   case (qid, abind(qid1, aval, aml)) =>
186     if (qid == qid1)
187       someAval(aval)
188     else
189       lookupAnsMap(qid, aml)
190 }
191
192 def appendAnsMap(am1: AnsMap, am2: AnsMap): AnsMap = (am1, am2) match {
193   case (aempty(), am) => am
194   case (abind(qid, av, am), aml) => abind(qid, av, appendAnsMap(am, aml))
195 }
196
197 sealed trait QMap extends Expression
198 case class qmempty() extends QMap
199 case class qmbind(qid: QID, l: Label, atype: AType, qml: QMap) extends QMap
200
201 @FailableType
202 sealed trait OptQuestion
```

```
203 case class noQuestion() extends OptQuestion
204 case class someQuestion(qid: QID, l: Label, atype: AType) extends OptQuestion
205
206 def isSomeQuestion(opt: OptQuestion): Boolean = opt match {
207   case noQuestion() => false
208   case someQuestion(_, _, _) => true
209 }
210
211 @Partial
212 def getQuestionQID(opt: OptQuestion): QID = opt match {
213   case someQuestion(qid, l, at) => qid
214 }
215
216 @Partial
217 def getQuestionLabel(opt: OptQuestion): Label = opt match {
218   case someQuestion(qid, l, at) => l
219 }
220
221 @Partial
222 def getQuestionAType(opt: OptQuestion): AType = opt match {
223   case someQuestion(qid, l, at) => at
224 }
225
226 @Dynamic
227 @ProgressProperty("lookupQMapProgress")
228 @PreservationProperty("lookupQMapPreservation")
229 @Recursive(1)
230 def lookupQMap(id: QID, qm: QMap): OptQuestion = (id, qm) match {
231   case (_, qmempty()) => noQuestion()
232   case (qid, qmbind(qid1, l, at, qml)) =>
233     if (qid == qid1)
234       someQuestion(qid, l, at)
235     else
236       lookupQMap(qid, qml)
237 }
238
239 sealed trait QConf extends Expression
240 case class QC(am: AnsMap, qm: QMap, q: Questionnaire) extends QConf
241
242 def getAM(qc: QConf): AnsMap = qc match {
243   case QC(am, _, _) => am
244 }
245
246 def getQM(qc: QConf): QMap = qc match {
247   case QC(_, qm, _) => qm
248 }
249
250 def getQuest(qc: QConf): Questionnaire = qc match {
251   case QC(_, _, q) => q
252 }
253
```

```
254  def isValue(q: Questionnaire): Boolean = q match {
255    case qempty() => true
256    case _ => false
257  }
258
259  @FailableType
260  sealed trait OptQConf
261  case class noQConf() extends OptQConf
262  case class someQConf(qc: QConf) extends OptQConf
263
264  def isSomeQC(opt: OptQConf): Boolean = opt match {
265    case noQConf() => false
266    case someQConf(_) => true
267  }
268
269  @Partial
270  def getQC(opt: OptQConf): QConf = opt match {
271    case someQConf(qc) => qc
272  }
273
274  def qcappend(qc: QConf, q: Questionnaire): QConf = (qc, q) match {
275    case (QC(am, qm, qs1), qs2) => QC(am, qm, qseq(qs1, qs2))
276  }
277
278  @FailableType
279  sealed trait OptExp
280  case class noExp() extends OptExp
281  case class someExp(exp: Exp) extends OptExp
282
283  def isSomeExp(opt: OptExp): Boolean = opt match {
284    case noExp() => false
285    case someExp(_) => true
286  }
287
288  @Partial
289  def getExp(opt: OptExp): Exp = opt match {
290    case someExp(e) => e
291  }
292
293  def expIsValue(exp: Exp): Boolean = exp match {
294    case constant(_) => true
295    case e => false
296  }
297
298  @Partial
299  def getExpValue(exp: Exp): Aval = exp match {
300    case constant(av) => av
301  }
302
303  // QLSemantics
304
```

```
305  def askYesNo(l: Label): YN = ???
306  def askNumber(l: Label): nat = ???
307  def askText(l: Label): string = ???
308
309  def getAnswer(l: Label, at: AType): Aval = (l, at) match {
310    case (l, YesNo()) => B(askYesNo(l))
311    case (l, Number()) => Num(askNumber(l))
312    case (l, Text()) => T(askText(l))
313  }
314
315  @Dynamic
316  @ProgressProperty("evalBinOpProgress")
317  @PreservationProperty("evalBinOpPreservation")
318  @TopLevelDistinctionHint(0, 1, 2)
319  def evalBinOp(op: BinOpT, av1: Aval, av2: Aval): OptExp = (op, av1, av2) match {
320    case (addop(), Num(n1), Num(n2)) => someExp(constant(Num(plus(n1, n2))))
321    case (subop(), Num(n1), Num(n2)) => someExp(constant(Num(minus(n1, n2))))
322    case (mulop(), Num(n1), Num(n2)) => someExp(constant(Num(multiply(n1, n2))))
323    case (divop(), Num(n1), Num(n2)) => someExp(constant(Num(divide(n1, n2))))
324    case (gtop(), Num(n1), Num(n2)) => someExp(constant(B(gt(n1, n2))))
325    case (ltop(), Num(n1), Num(n2)) => someExp(constant(B(lt(n1, n2))))
326    case (andop(), B(b1), B(b2)) => someExp(constant(B(and(b1, b2))))
327    case (orop(), B(b1), B(b2)) => someExp(constant(B(or(b1, b2))))
328    case (eqop(), a, a1) =>
329      if(a == a1)
330        someExp(constant(B(yes())))
331      else
332        someExp(constant(B(no())))
333    case (_, _, _) => noExp()
334  }
335
336  @Dynamic
337  @ProgressProperty("evalUnOpProgress")
338  @PreservationProperty("evalUnOpPreservation")
339  def evalUnOp(op: UnOpT, av: Aval): OptExp = (op, av) match {
340    case (notop(), B(b)) => someExp(constant(B(not(b))))
341    case (_, _) => noExp()
342  }
343
344  @Dynamic
345  @ProgressProperty("reduceExpProgress")
346  @PreservationProperty("reduceExpPreservation")
347  @Recursive(0)
348  def reduceExp(exp: Exp, amap: AnsMap): OptExp = (exp, amap) match {
349    case (constant(av), _) => noExp()
350    case (qvar(qid), am) =>
351      val avOpt = lookupAnsMap(qid, am)
352      if (isSomeAval(avOpt))
353        someExp(constant(getAval(avOpt)))
354      else
355        noExp()
```

```
356     case (binop(e1, op, e2), am) =>
357       if (expIsValue(e1))
358         if (expIsValue(e2))
359           evalBinOp(op, getExpValue(e1), getExpValue(e2))
360         else {
361           val eOpt2 = reduceExp(e2, am)
362           if (isSomeExp(eOpt2))
363             someExp(binop(e1, op, getExp(eOpt2)))
364           else noExp()
365         }
366       else {
367         val eOpt1 = reduceExp(e1, am)
368         if (isSomeExp(eOpt1))
369           someExp(binop(getExp(eOpt1), op, e2))
370         else noExp()
371       }
372     case (unop(op, e1), am) =>
373       if (expIsValue(e1))
374         evalUnOp(op, getExpValue(e1))
375       else {
376         val eOpt = reduceExp(e1, am)
377         if (isSomeExp(eOpt))
378           someExp(unop(op, getExp(eOpt)))
379         else noExp()
380       }
381   }
382
383   @Recursive(0)
384   @ProgressProperty("Progress")
385   @PreservationProperty("Preservation")
386   @Dynamic
387   def reduce(q: Questionnaire, ama: AnsMap, qma: QMap): OptQConf = (q, ama, qma)
          match {
388     case (qempty(), _, _) => noQConf()
389     case (qsingle(question(qid, l, t)), am, qm) =>
390       val av = getAnswer(l, t)
391       someQConf(QC(abind(qid, av, am), qm, qempty()))
392     case (qsingle(value(qid, t, exp)), am, qm) =>
393       if (expIsValue(exp))
394         someQConf(QC(abind(qid, getExpValue(exp), am), qm, qempty()))
395       else {
396         val eOpt = reduceExp(exp, am)
397         if (isSomeExp(eOpt))
398           someQConf(QC(am, qm, qsingle(value(qid, t, getExp(eOpt)))))
399         else noQConf()
400       }
401     case (qsingle(defquestion(qid, l, t)), am, qm) =>
402       someQConf(QC(am, qmbind(qid, l, t, qm), qempty()))
403     case (qsingle(ask(qid)), am, qm) =>
404       val qOpt = lookupQMap(qid, qm)
405       if (isSomeQuestion(qOpt))
```

```
406         someQConf(QC(am, qm, qsingle(question(qid,
407           getQuestionLabel(qOpt),
408           getQuestionAType(qOpt)))))
409       else noQConf()
410     case (qseq(qempty(), qs), am, qm) => someQConf(QC(am, qm, qs))
411     case (qseq(qs1, qs2), am, qm) =>
412       val qcOpt = reduce(qs1, am, qm)
413       if (isSomeQC(qcOpt))
414         someQConf(qcappend(getQC(qcOpt), qs2))
415       else noQConf()
416     case (qcond(constant(B(yes())), qs1, qs2), am, qm) => someQConf(QC(am, qm, qs1))
417     case (qcond(constant(B(no())), qs1, qs2), am, qm) => someQConf(QC(am, qm, qs2))
418     case (qcond(e, qs1, qs2), am, qm) =>
419       val eOpt = reduceExp(e, am)
420       if (isSomeExp(eOpt))
421         someQConf(QC(am, qm, qcond(getExp(eOpt), qs1, qs2)))
422       else noQConf()
423     case (qgroup(_, qs), am, qm) => someQConf(QC(am, qm, qs))
424   }
425
426   // QLTypeSystem
427
428   sealed trait ATMap extends Context with Type
429   case class atmempty() extends ATMap
430   case class atmbind(qid: QID, at: AType, atml: ATMap) extends ATMap
431
432   @Static
433   @Recursive(1)
434   def lookupATMap(qid: QID, atm: ATMap): OptAType = (qid, atm) match {
435     case (_, atmempty()) => noAType()
436     case (qid1, atmbind(qid2, at, atml)) =>
437       if (qid1 == qid2) someAType(at)
438       else lookupATMap(qid1, atml)
439   }
440
441   @Static
442   @Recursive(0)
443   def appendATMap(atm1: ATMap, atm2: ATMap): ATMap = (atm1, atm2) match {
444     case (atmempty(), atm) => atm
445     case (atmbind(qid, at, atm), atml) => atmbind(qid, at, appendATMap(atm, atml))
446   }
447
448   @Static
449   @Recursive(0)
450   def intersectATM(atm1: ATMap, atm2: ATMap): ATMap = (atm1, atm2) match {
451     case (atmempty(), _) => atmempty()
452     //case (_, atmempty()) => atmempty() //shortcut case is a bit problematic with
          current ACG construction, maybe fix later
453     case (atmbind(qid, at, atm1), atm2) =>
454       val atm1atm2 = intersectATM(atm1, atm2)
455       val lAT = lookupATMap(qid, atm2)
```

```
456       if (isSomeAType(lAT) && getAType(lAT) == at)
457         atmbind(qid, at, atm1atm2)
458       else
459         atm1atm2
460   }
461
462   sealed trait MapConf extends Context with Type
463   case class MC(atm: ATMap, qtm: ATMap) extends MapConf
464
465   @FailableType
466   sealed trait OptMapConf
467   case class noMapConf() extends OptMapConf
468   case class someMapConf(mc: MapConf) extends OptMapConf
469
470   def isSomeMapConf(opt: OptMapConf): Boolean = opt match {
471     case noMapConf() => false
472     case someMapConf(_) => true
473   }
474
475   @Partial
476   def getMapConf(opt: OptMapConf): MapConf = opt match {
477     case someMapConf(mc) => mc
478   }
479
480   @Static
481   @Recursive(0)
482   def typeAM(am: AnsMap): ATMap = am match {
483     case aempty() => atmempty()
484     case abind(qid, av, amr) => atmbind(qid, typeOf(av), typeAM(amr))
485   }
486
487   @Static
488   @Recursive(0)
489   def typeQM(qm: QMap): ATMap = qm match {
490     case qmempty() => atmempty()
491     case qmbind(qid, _, at, qmr) => atmbind(qid, at, typeQM(qmr))
492   }
493
494   def checkBinOp(op: BinOpT, at1: AType, at2: AType): OptAType = (op, at1, at2) match
          {
495     case (addop(), Number(), Number()) => someAType(Number())
496     case (subop(), Number(), Number()) => someAType(Number())
497     case (mulop(), Number(), Number()) => someAType(Number())
498     case (divop(), Number(), Number()) => someAType(Number())
499     case (gtop(), Number(), Number()) => someAType(YesNo())
500     case (ltop(), Number(), Number()) => someAType(YesNo())
501     case (andop(), YesNo(), YesNo()) => someAType(YesNo())
502     case (orop(), YesNo(), YesNo()) => someAType(YesNo())
503     case (eqop(), _, _) => someAType(YesNo())
504     case (_, _, _) => noAType()
505   }
```

```
506
507   @Static
508   def checkUnOp(op: UnOpT, at: AType): OptAType = (op, at) match {
509     case (notop(), YesNo()) => someAType(YesNo())
510     case (_, _) => noAType()
511   }
512
513   @Static
514   @Recursive(1)
515   def echeck(atmap: ATMap, exp: Exp): OptAType = (atmap, exp) match {
516     case (_, constant(B(n))) => someAType(YesNo())
517     case (_, constant(Num(n))) => someAType(Number())
518     case (_, constant(T(n))) => someAType(Text())
519     case (atm, qvar(qid)) => lookupATMap(qid, atm)
520     case (atm, binop(e1, op, e2)) =>
521       val t1 = echeck(atm, e1)
522       val t2 = echeck(atm, e2)
523       if(isSomeAType(t1) && isSomeAType(t2))
524         checkBinOp(op, getAType(t1), getAType(t2))
525       else noAType()
526     case (atm, unop(op, e)) =>
527       val t = echeck(atm, e)
528       if (isSomeAType(t))
529         checkUnOp(op, getAType(t))
530       else noAType()
531   }
532
533   @Axiom
534   def Tqempty(atm: ATMap, qm: ATMap): Unit = {
535   } ensuring MC(atm, qm) |- qempty() :: MC(atm, qm)
536
537   @Axiom
538   def Tquestion(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType): Unit = {
539     require(lookupATMap(qid, atm) == noAType())
540   } ensuring MC(atm, qm) |- qsingle(question(qid, l, at)) :: MC(atmbind(qid, at, atm), qm)
541
542   @Axiom
543   def Tvalue(qid: QID, atm: ATMap, exp: Exp, qm: ATMap, at: AType): Unit = {
544     require(lookupATMap(qid, atm) == noAType())
545     require(echeck(atm, exp) == someAType(at))
546   } ensuring MC(atm, qm) |- qsingle(value(qid, at, exp)) :: MC(atmbind(qid, at, atm), qm)
547
548   @Axiom
549   def Tdefquestion(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType): Unit = {
550     require(lookupATMap(qid, qm) == noAType())
551   } ensuring MC(atm, qm) |- qsingle(defquestion(qid, l, at)) :: MC(atm, atmbind(qid, at,
        qm))
552
553   @Axiom
554   def Task(qid: QID, qm: ATMap, at: AType, atm: ATMap): Unit = {
555     require(lookupATMap(qid, qm) == someAType(at))
```

```
556     require(lookupATMap(qid, atm) == noAType())
557   } ensuring MC(atm, qm) |− qsingle(ask(qid)) :: MC(atmbind(qid, at, atm), qm)
558
559   @Axiom
560   def Tqseq(atm: ATMap, qm: ATMap, q1: Questionnaire, atm1: ATMap,
561       atm2: ATMap, qm1: ATMap, q2: Questionnaire, qm2: ATMap): Unit = {
562     require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
563     require(MC(atm1, qm1) |− q2 :: MC(atm2, qm2))
564   } ensuring MC(atm, qm) |− qseq(q1, q2) :: MC(atm2, qm2)
565
566   @Axiom
567   def Tqcond(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
568       atm1: ATMap, qm1: ATMap, q2: Questionnaire): Unit = {
569     require(echeck(atm, exp) == someAType(YesNo()))
570     require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
571     require(MC(atm, qm) |− q2 :: MC(atm1, qm1))
572   } ensuring MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atm1, qm1)
573
574   @Axiom
575   def Tqgroup(atm: ATMap, qm: ATMap, q: Questionnaire, atm1: ATMap, qm1: ATMap,
        gid: GID): Unit = {
576     require(MC(atm, qm) |− q :: MC(atm1, qm1))
577   } ensuring (MC(atm, qm) |− qgroup(gid, q) :: MC(atm1, qm1))
578
579
580   //type inversion axioms
581   @Axiom
582   def Tqempty_inv1(atm: ATMap, qm: ATMap, atm1: ATMap, qm1: ATMap): Unit = {
583     require (MC(atm, qm) |− qempty() :: MC(atm1, qm1))
584   } ensuring (atm1 == atm)
585
586   @Axiom
587   def Tqempty_inv2(atm: ATMap, qm: ATMap, atm1: ATMap, qm1: ATMap): Unit = {
588     require (MC(atm, qm) |− qempty() :: MC(atm1, qm1))
589   } ensuring (qm1 == qm)
590
591
592   @Axiom
593   def Tquestion_inv1(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType, atm1:
        ATMap, qm1: ATMap): Unit = {
594     require(MC(atm, qm) |− qsingle(question(qid, l, at)) :: MC(atm1, qm1))
595   } ensuring (lookupATMap(qid, atm) == noAType())
596
597   @Axiom
598   def Tquestion_inv2(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType, atm1:
        ATMap, qm1: ATMap): Unit = {
599     require(MC(atm, qm) |− qsingle(question(qid, l, at)) :: MC(atm1, qm1))
600   } ensuring (atm1 == atmbind(qid, at, atm))
601
602   @Axiom
```

```
603   def Tquestion_inv3(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType, atm1:
          ATMap, qm1: ATMap): Unit = {
604     require(MC(atm, qm) |− qsingle(question(qid, l, at)) :: MC(atm1, qm1))
605   } ensuring (qm1 == qm)
606
607
608   @Axiom
609   def Tvalue_inv1(qid: QID, atm: ATMap, exp: Exp, qm: ATMap, at: AType, atm1: ATMap,
          qm1: ATMap): Unit = {
610     require(MC(atm, qm) |− qsingle(value(qid, at, exp)) :: MC(atm1, qm1))
611   } ensuring (lookupATMap(qid, atm) == noAType())
612
613   @Axiom
614   def Tvalue_inv2(qid: QID, atm: ATMap, exp: Exp, qm: ATMap, at: AType, atm1: ATMap,
          qm1: ATMap): Unit = {
615     require(MC(atm, qm) |− qsingle(value(qid, at, exp)) :: MC(atm1, qm1))
616   } ensuring (echeck(atm, exp) == someAType(at))
617
618   @Axiom
619   def Tvalue_inv3(qid: QID, atm: ATMap, exp: Exp, qm: ATMap, at: AType, atm1: ATMap,
          qm1: ATMap): Unit = {
620     require(MC(atm, qm) |− qsingle(value(qid, at, exp)) :: MC(atm1, qm1))
621   } ensuring (atm1 == atmbind(qid, at, atm))
622
623   @Axiom
624   def Tvalue_inv4(qid: QID, atm: ATMap, exp: Exp, qm: ATMap, at: AType, atm1: ATMap,
          qm1: ATMap): Unit = {
625     require(MC(atm, qm) |− qsingle(value(qid, at, exp)) :: MC(atm1, qm1))
626   } ensuring (qm1 == qm)
627
628
629   @Axiom
630   def Tdefquestion_inv1(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType, atm1:
          ATMap, qm1: ATMap): Unit = {
631     require (MC(atm, qm) |− qsingle(defquestion(qid, l, at)) :: MC(atm1, qm1))
632   } ensuring (lookupATMap(qid, qm) == noAType())
633
634   @Axiom
635   def Tdefquestion_inv2(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType, atm1:
          ATMap, qm1: ATMap): Unit = {
636     require (MC(atm, qm) |− qsingle(defquestion(qid, l, at)) :: MC(atm1, qm1))
637   } ensuring (atm1 == atm)
638
639   @Axiom
640   def Tdefquestion_inv3(qid: QID, atm: ATMap, qm: ATMap, l: Label, at: AType, atm1:
          ATMap, qm1: ATMap): Unit = {
641     require (MC(atm, qm) |− qsingle(defquestion(qid, l, at)) :: MC(atm1, qm1))
642   } ensuring (qm1 == atmbind(qid, at, qm))
643
644
645   @Axiom
```

```
646   def Task_inv1(qid: QID, qm: ATMap, at: AType, atm: ATMap, atm1: ATMap, qm1:
          ATMap): Unit = {
647     require(MC(atm, qm) |− qsingle(ask(qid)) :: MC(atm1, qm1))
648   } ensuring (lookupATMap(qid, atm) == noAType())
649
650   @Axiom
651   def Task_inv2(qid: QID, qm: ATMap, atm: ATMap, atm1: ATMap, qm1: ATMap): Unit =
          {
652     require(MC(atm, qm) |− qsingle(ask(qid)) :: MC(atm1, qm1))
653   } ensuring (exists ((at: AType) => lookupATMap(qid, qm) == someAType(at)))
654
655   @Axiom
656   def Task_inv3(qid: QID, qm: ATMap, at: AType, atm: ATMap, atm1: ATMap, qm1:
          ATMap): Unit = {
657     require(MC(atm, qm) |− qsingle(ask(qid)) :: MC(atm1, qm1))
658     require(lookupATMap(qid, qm) == someAType(at))
659   } ensuring (atm1 == atmbind(qid, at, atm))
660
661   @Axiom
662   def Task_inv4(qid: QID, qm: ATMap, atm: ATMap, atm1: ATMap, qm1: ATMap): Unit =
          {
663     require(MC(atm, qm) |− qsingle(ask(qid)) :: MC(atm1, qm1))
664   } ensuring (qm1 == qm)
665
666
667   @Axiom
668   def Tqseq_inv1(atm: ATMap, qm: ATMap, q1: Questionnaire, q2: Questionnaire, atmr:
          ATMap, qmr: ATMap): Unit = {
669     require(MC(atm, qm) |− qseq(q1, q2) :: MC(atmr, qmr))
670   } ensuring (exists ((atm1: ATMap, qm1: ATMap) => MC(atm, qm) |− q1 :: MC(atm1,
          qm1)))
671
672   @Axiom
673   def Tqseq_inv2(atm: ATMap, qm: ATMap, q1: Questionnaire, q2: Questionnaire, atmr:
          ATMap, qmr: ATMap, atm1: ATMap, qm1: ATMap): Unit = {
674     require(MC(atm, qm) |− qseq(q1, q2) :: MC(atmr, qmr))
675     require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
676   } ensuring (exists ((atm2: ATMap, qm2: ATMap) => MC(atm1, qm1) |− q2 :: MC(atm2,
          qm2)))
677
678   @Axiom
679   def Tqseq_inv3(atm: ATMap, qm: ATMap, q1: Questionnaire, q2: Questionnaire, atmr:
          ATMap, qmr: ATMap, atm1: ATMap, qm1: ATMap, atm2: ATMap, qm2: ATMap):
          Unit = {
680     require(MC(atm, qm) |− qseq(q1, q2) :: MC(atmr, qmr))
681     require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
682     require(MC(atm1, qm1) |− q2 :: MC(atm2, qm2))
683   } ensuring (atmr == atm2)
684
685   @Axiom
```

```
686  def Tqseq_inv4(atm: ATMap, qm: ATMap, q1: Questionnaire, q2: Questionnaire, atmr:
         ATMap, qmr: ATMap, atm1: ATMap, qm1: ATMap, atm2: ATMap, qm2: ATMap):
         Unit = {
687    require(MC(atm, qm) |− qseq(q1, q2) :: MC(atmr, qmr))
688    require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
689    require(MC(atm1, qm1) |− q2 :: MC(atm2, qm2))
690  } ensuring (qmr == qm2)
691
692
693  @Axiom
694  def Tqcond_inv1(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
695              q2: Questionnaire, atmr: ATMap, qmr: ATMap): Unit = {
696    require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
697  } ensuring (echeck(atm, exp) == someAType(YesNo()))
698
699  @Axiom
700  def Tqcond_inv2(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
701                  q2: Questionnaire, atmr: ATMap, qmr: ATMap): Unit = {
702    require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
703  } ensuring (exists ((atm1: ATMap, qm1: ATMap) => MC(atm, qm) |− q1 :: MC(atm1,
         qm1)))
704
705  @Axiom
706  def Tqcond_inv3(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
707                  q2: Questionnaire, atmr: ATMap, qmr: ATMap): Unit = {
708    require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
709  } ensuring (exists ((atm1: ATMap, qm1: ATMap) => MC(atm, qm) |− q2 :: MC(atm1,
         qm1)))
710
711
712  @Axiom
713  def Tqcond_inv4(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
714                  q2: Questionnaire, atmr: ATMap, qmr: ATMap, atm1: ATMap, qm1:
                        ATMap, atm2: ATMap, qm2: ATMap): Unit = {
715    require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
716    require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
717    require(MC(atm, qm) |− q2 :: MC(atm2, qm2))
718  } ensuring (atm1 == atm2)
719
720  @Axiom
721  def Tqcond_inv5(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
722                  q2: Questionnaire, atmr: ATMap, qmr: ATMap, atm1: ATMap, qm1:
                        ATMap, atm2: ATMap, qm2: ATMap): Unit = {
723    require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
724    require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
725    require(MC(atm, qm) |− q2 :: MC(atm2, qm2))
726  } ensuring (qm1 == qm2)
727
728  @Axiom
729  def Tqcond_inv6(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
```

```
730                     q2: Questionnaire, atmr: ATMap, qmr: ATMap, atm1: ATMap, qm1:
                            ATMap): Unit = {
731     require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
732     require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
733     require(MC(atm, qm) |− q2 :: MC(atm1, qm1))
734   } ensuring (atmr == atm1)
735
736   @Axiom
737   def Tqcond_inv7(atm: ATMap, exp: Exp, qm: ATMap, q1: Questionnaire,
738                     q2: Questionnaire, atmr: ATMap, qmr: ATMap, atm1: ATMap, qm1:
                            ATMap): Unit = {
739     require(MC(atm, qm) |− qcond(exp, q1, q2) :: MC(atmr, qmr))
740     require(MC(atm, qm) |− q1 :: MC(atm1, qm1))
741     require(MC(atm, qm) |− q2 :: MC(atm1, qm1))
742   } ensuring (qmr == qm1)
743
744
745
746   @Axiom
747   def Tqgroup_inv(atm: ATMap, qm: ATMap, q: Questionnaire, atm1: ATMap, qm1:
          ATMap, gid: GID): Unit = {
748     require(MC(atm, qm) |− qgroup(gid, q) :: MC(atm1, qm1))
749   } ensuring (MC(atm, qm) |− q :: MC(atm1, qm1))
750
751
752
753
754   //Progress property and necessary auxiliary lemmas
755   @Property
756   def Progress(am: AnsMap, qm: QMap, q: Questionnaire, atm: ATMap,
757       qtm: ATMap, atm2: ATMap, qtm2: ATMap): Unit = {
758     require(!isValue(q))
759     require(typeAM(am) == atm)
760     require(typeQM(qm) == qtm)
761     require(MC(atm, qtm) |− q :: MC(atm2, qtm2))
762   } ensuring exists((am0: AnsMap, qm0: QMap, q0: Questionnaire) =>
763     reduce(q, am, qm) == someQConf(QC(am0, qm0, q0)))
764
765   @Property
766   def reduceExpProgress(e: Exp, am: AnsMap, atm: ATMap, at: AType): Unit = {
767     require(!expIsValue(e))
768     require(typeAM(am) == atm)
769     require(echeck(atm, e) == someAType(at))
770   } ensuring exists((eres: Exp) => reduceExp(e, am) == someExp(eres))
771
772   @Property
773   def lookupAnsMapProgress(am: AnsMap, atm: ATMap, qid: QID, at: AType): Unit = {
774     require(typeAM(am) == atm)
775     require(lookupATMap(qid, atm) == someAType(at))
776   } ensuring exists((av0: Aval) => lookupAnsMap(qid, am) == someAval(av0))
777
```

```
778  @Property
779  def lookupQMapProgress(qm: QMap, qtm: ATMap, qid: QID, at: AType): Unit = {
780    require(typeQM(qm) == qtm)
781    require(lookupATMap(qid, qtm) == someAType(at))
782  } ensuring exists((qid0: QID, l0: Label, t0: AType) =>
783    lookupQMap(qid, qm) == someQuestion(qid0, l0, t0))
784
785  @Property
786  def evalBinOpProgress(atm: ATMap, bot: BinOpT, at: AType, a: Aval, a1: Aval): Unit = {
787    require(echeck(atm, binop(constant(a), bot, constant(a1))) == someAType(at))
788  } ensuring(exists ((eres: Exp) => evalBinOp(bot, a, a1) == someExp(eres)))
789
790  @Property
791  def evalUnOpProgress(atm: ATMap, uot: UnOpT, at: AType, a: Aval, a1: Aval): Unit = {
792    require(echeck(atm, unop(uot, constant(a))) == someAType(at))
793  } ensuring(exists ((eres: Exp) => evalUnOp(uot, a) == someExp(eres)))
794
795
796  //Preservation property and necessary auxiliary lemmas
797  @Property
798  def Preservation(atm: ATMap, qtm: ATMap, q: Questionnaire, atm1: ATMap, qtm1:
         ATMap,
799                   am: AnsMap, qm: QMap, amr: AnsMap, qmr: QMap, qr: Questionnaire,
                         atmr: ATMap, qtmr: ATMap): Unit = {
800    require(MC(atm, qtm) |- q :: MC(atm1, qtm1))
801    require(typeAM(am) == atm)
802    require(typeQM(qm) == qtm)
803    require(reduce(q, am, qm) == someQConf(QC(amr, qmr, qr)))
804    require(atmr == typeAM(amr))
805    require(qtmr == typeQM(qmr))
806  } ensuring (MC(atmr, qtmr) |- qr :: MC(atm1, qtm1))
807
808  @Property
809  def reduceExpPreservation(e: Exp, am: AnsMap, atm: ATMap, at: AType, er: Exp): Unit
         = {
810    require(echeck(atm, e) == someAType(at))
811    require(typeAM(am) == atm)
812    require(reduceExp(e, am) == someExp(er))
813  } ensuring(echeck(atm, er) == someAType(at))
814
815  @Property
816  def lookupAnsMapPreservation(am: AnsMap, atm: ATMap, qid: QID, at: AType, avr:
         Aval): Unit = {
817    require(lookupATMap(qid, atm) == someAType(at))
818    require(lookupAnsMap(qid, am) == someAval(avr))
819    require(typeAM(am) == atm)
820  } ensuring(typeOf(avr) == at)
821
822  @Property
823  def lookupQMapPreservation(qm: QMap, atm: ATMap, qid: QID, at: AType, l: Label, t:
         AType): Unit = {
```

```
824     require(lookupATMap(qid, atm) == someAType(at))
825     require(lookupQMap(qid, qm) == someQuestion(qid, l, t))
826     require(typeQM(qm) == atm)
827   } ensuring(at == t)
828
829   @Property
830   def evalBinOpPreservation(atm: ATMap, bot: BinOpT, at: AType, a: Aval, a1: Aval, eres:
          Exp): Unit = {
831     require(echeck(atm, binop(constant(a), bot, constant(a1))) == someAType(at))
832     require(evalBinOp(bot, a, a1) == someExp(eres))
833   } ensuring(echeck(atm, eres) == someAType(at))
834
835   @Property
836   def evalUnOpPreservation(uot: UnOpT, av: Aval, atm: ATMap, at: AType, eres: Exp):
          Unit = {
837     require(echeck(atm, unop(uot, constant(av))) == someAType(at))
838     require(evalUnOp(uot, av) == someExp(eres))
839   } ensuring(echeck(atm, eres) == someAType(at))
840
841 }
```

Listing A.3: Full specification of typed QL case study in ScalaSPL

List of Figures

List of Tables

List of Listings

List of Definitions and Theorems

Index